**MASTER EN SCIENCES INFORMATIQUES**

# The kernel of a real-time monitor for multiple processor microcomputer systems

Wautelet, Christian

*Award date:*
1978

*Awarding institution:*
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE - DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1977 - 1978

# THE KERNEL OF A REAL - TIME MONITOR FOR MULTIPLE PROCESSOR MICROCOMPUTER SYSTEMS

CHRISTIAN WAUTELET

Mémoire présenté en vue de
l'obtention du grade de
Licencié et Maître en Informatique.

FACULTES UNIVERSITAIRES NOTRE - DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1977 - 1978

# THE KERNEL OF A REAL - TIME MONITOR FOR MULTIPLE PROCESSOR MICROCOMPUTER SYSTEMS

CHRISTIAN WAUTELET

Mémoire présenté en vue de
l'obtention du grade de
Licencié et Maître en Informatique.

Acknowledgements.

My deep gratitude for the help and advise I have received from :

- Mr. Stéphane de Epsée : thesis director
- Mr. Joseph Demarteau : thesis advisor.

I also wish to thank Mrs Christiane Lemoine for the dedication with which she typed this text.

C O N T E N T S

# I N T R O D U C T I O N

## Multi-microcomputer systems

The low cost of microprocessors allows systems application engineers to think in terms of multi-microcomputers systems. One of the advantages is to enhance system performances. This can be achieved, on the one hand, by sectioning the tasks that must be performed by the system, into functions that can be handle by individual processes, and on the other hand, by distributing these processes among the different microcomputers so that they can work simultaneously and accomplish a same global task.

These facilities will be particularly appreciated in the design of real-time applications, because a well-planned distribution of processes among the processors, will permit a reduction of response times to service requests. In particular, the interrupt signals could be treated by a set of specialized processors rather than by one processor only, as it is in the case of mono-processor systems.

Therefore, it is reasonnable to think that multi-microcomputer systems will increase considerably in the next few years and that the needs of multiprocessor application specialists will follow that tendency. However, the complexity of the problems inherent to the utilisation of such systems (in particular, the software problems) leads us to believe that to overcome them will need a certain amount of time.

## Objectives

The purpose of this thesis has been to acquire a solid basis which could lead to the field of multi-microprocessor applications. The two objectives aimed at attaining this purpose are :

- The acquisition of a theoretical knowledge of the main concepts that are inherent to the multiprocesses and multiprocessors environments.

- The consolidation of knowledge by defining the kernel of a real-time monitor that could be easily implemented and used for a certain type of multi-microcomputer applications.

General approach

Therefore, two steps were necessary to achieve this purpose : The first step has been an important bibliographical work, the purpose of which was to acquire the main concepts of multiprocesses environments, and to select among these the ones that seemed to be realisable without appealing to powerful (and costly) data processing systems.
The second step was to define a standard multiple microcomputer configuration and construct the kernel of a real time monitor that could be used for such a configuration.

General Philosophy

From this works' theoretical aspects, a general philosophy appeared and may be summarized as follows :

1. A net distinction is done between the notions of monitor and operating system :
   - An operating system is defined as a set of processes which have to schedule and control the activities of other processes.
   - A monitor, is a set of data and procedures created to help the processes to respect the system utilisation rules defined by the system designer. A monitor is then a set of tools geared for the process disposal.

2. This Monitor definition can free us from the constraint of submitting the application processes to an operating system created to satisfy the requirements of all the other processes (which have very often totally different targets).Such a system could be heavy, and would consequently reduce the flexibility

of the multi-microcomputer system. Therefore, to keep a maximum of flexibility, we will suppress, at the monitor level, all distinction between an operating system and user processes and say that the monitor procedures may be accessible by all the processes and not only by a part of them.

3. The monitor procedures must be defined in a modular fashion, in such a way that it must be easy to modify them with regard to the applications. In other words, we want to be able to adapt a monitor to the applications that the system must process, and not necessarily the applications to a standard monitor.

Structure of Thesis

This thesis includes two chapters :

- The first chapter is a synthesis of the main concepts, found during the bibliographical study and that will be needed for the monitor described in the next chapter.

These concepts may be summarized by the following terms :

- processes and states of processes
- processes synchronization
- mutual exclusion
- deadlock
- interprocess communication

The second chapter describes a monitor for applications in which the programs are fixed, once and for all, in private ROM memories, and where interprocess communication and synchronization are done essentially through the use of a common memory accessible by all the processors. This monitor has been written (with a lot of comments) in a structured programmation language (Pascal). This language is used only as a conceptual language, and the procedures are sufficiently detailed to be easily translated into one (or more) assembly language(s) for a final implementation.

Next step

The next step, in course of realisation, consists in translating the kernel above defined into the Intel 8080 Assembly language, with the purpose of implementing it for applications using the Intel multibus, on which can be connected single-board microcomputers of the SBC 80 family.

NOTE

This thesis has been written in English for practical reasons, one of which being the use as basic documentation for future implementation.

I am grateful to the University Notre-Dame-de-la-Paix for having allowed me to write it in that language.

# CHAPTER 1 : Generalities

## 1.1. Processes

We define a process as an entity which has to execute a program on a processor.

In a multiprocesses environment, a process can take 3 states - (We suppose that the number of processors is less than the number of processes) :

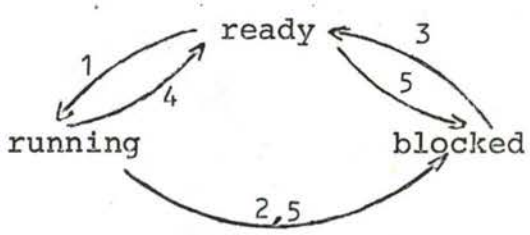
- running : if it is executing a program on a processor.
- ready : if it is waiting for the disponibility of a processor to execute its program.
- blocked : if it is waiting outside a processor for a signal other than the disponibility of a processor.

To change the state of a process, the following rules must be respected :

1. Only ready processes can be made running.
2. When a running process must block, it leaves its processor.
3. When a signal arrives to wake up a blocked process, the process is made ready.
4. A running process can be preempted to the ready state, to give the control of the processor to another process.
5. A process can block itself or be blocked by another process.

NOTE : A process which is executing a waiting-loop is not blocked, but running.

The process states diagram may be represented as follows :




The digits in that diagram refer to the rules that we have defined.

Not running processes will normally wait in a queue of processes. As we will see later, a queue of processes will be defined as a sequence of process names, a process name being an information assigned to the process to identify it.
When a processor is free, it is given to a process waiting in a "ready queue".
When a signal wakes up a process, the process will have to leave the queue where it is waiting, to enter a ready queue before it can continue the execution of its program.
These operations and all operations on processes will be executed under control of the Monitor (see 1.2.).

## 1.2. Monitor Definition

We define a Monitor as a set of procedures and data that can be regarded as a "software extention of the hardware structure" (BH 1) to implement a set of rules that must be respected by the processes for a correct utilization of the system.

In that sense, a monitor is not a process but a set of tools used by the processes.

## 1.3. Processes Synchronization

### 1.3.1. Introduction

By processes synchronization we intend that a process must be able, on the one hand, to wait for the arrival of a signal sent to it by another process, and, on the other hand, to send such a signal to a process waiting for it and to awake that process if it is blocked.

All these operations will be done under control of the Monitor, by executing synchronization primitives.

We can define two general rules concerning processes synchronization :

1. When a process sends a signal, it must be sure that this signal will be received.
2. When a process blocks itself to wait for a signal, it must be assured that it will be awakened after a finite time.

We will see 4 synchronization mechanisms :

- synchronization by wake up-waiting switches.
- synchronization by events.
- synchronization by private semaphores.
- synchronization by public semaphores.

In those mechanisms, we will only consider processes that block themselves.

NOTE : A comparison of synchronizating tools is given at 1.3.6.

### 1.3.2. Synchronization by wake up-waiting switches (Sa, La2, Cr)

A wake up-waiting switch is a boolean variable associated to a process. It is given the value "TRUE" when a signal is sent to the process but the process is still running. When the process decides to wait for the arrival of such a signal, it first examines its wake up-waiting switch and will block itself only if the switch is off (has the value "FALSE").

The synchronization primitives may thus be defined as follows :

block : if wake up-waiting switch (running) = 0
then running process state := blocked
else wake up-waiting switch (running) := 0

(wake up-waiting switch (running) means : the wake up-waiting switch of the running process which executes the block operation).

wakeup(p) : if process state (p) = blocked
then process state (p) := ready
else wake up-waiting switch (p) := 1

(p is the name of the receiver process)

We must take care of the following rules before using a wake up-waiting switch mechanism :

1. Only one process (the wake up-waiting switch owner) can use the switch to block itself.
2. Any process which knows the name of process p can execute the wakeup(p) operation.

Remarks :

1. If more than one process execute the wakeup(p) operation before process p decides to examine its wake up-waiting switch, only one signal (the last one) will be taken into consideration ; the others will be lost.

If no signal must be lost, a private semaphore should be used (see 1.3.4.).

2. Any process which wants to send a signal to process p must know the name of that process, but there is no way for process p to know the identity of the sending process.

3. If there is only one wake up-waiting switch by process, there is no way for a receiver process to distinguish the nature of a signal sent to it.

### 1.3.3. Synchronization by events (Cr)

An event is a boolean variable associated to the nature of a signal (event) rather than to a process.
The main difference with a wake up-waiting switch is that it can be examined by any process which wants to wait for the arrival of the event, and not by one particular process only. For that reason, a queue of processes is generally associated to an event.

We must take care of the following problems prior to defining the event monitor procedures :

1. When an event occurs and the queue is not empty, should we wake up all the waiting processes or only one of them ?
2. When a process examines an event and finds it "TRUE", should the process switch off the event ? If it does, the event will be lost for the other processes. If it does not, there is a danger for the process to loop on a same occurrence of the event.

The first problem can be resolved by introducing the use of public semaphores (see 1.3.5.). By definition, a signal sent to a public semaphore awakes only one process waiting for it

(if there is such a process). For that reason, we will reserve the use of an event when we want to awake, at the same time, all the processes waiting for an arrival of a signal.

To respond to the other questions we will introduce into the monitor five event procedures :

1. The procedure "wait(event)" blocks the calling process only if the event has not occurred. It can be defined as follows :

   wait(event) : if event = 1 then event:= 0
   else block running process
   in the event waiting queue.

2. The procedure "block(event)" blocks the calling process even if the event has already occurred. It is definded as follows :

   block (event) : blocks the running process in the event waiting queue.

Processes could execute that procedure to avoid a loop on a same occurrence of the event.

NOTE : A process which has executed the block(event) procedure will be awakened by the next occurrence of the event, but when it executes the block(event) operation, there is nothing which can certify to it that the event it wants to ignore is really the same that the one by which it was awakened.

3. The procedure "signal(event)" wakes up all the processes which are waiting for the next occurrence of the event, and switches off the value of the event, so that the signal will be lost for the other processes.

It is defined as follows :

```
signal(event) : if event queue = empty
                then event:= true
                else  begin
                        event:= 0 ; wake up all
                        processes waiting in the
                        queue
                      end
```

4. The procedure "awake(event)" awakes all the processes which are waiting in the event waiting queue and switches on the value of that event, so that the event will not be lost for the other processes.
It is defined as follows :

```
awake(event): event:=1;
              if eventqueue > <empty
              then wake up all processes waiting
                   in the queue
```

5. The procedure "reset(event)" gives the value "false" to the given event. It can be used by a privileged process to switch off the event when the awake(event) procedure is used to signal an occurrence of the event.

Remarks :

1. An event can only memorize one occurrence of an event signal.
2. It is possible to combine some events to define OR or AND functions.

### 1.3.4. Synchronization by private semaphores

A private semaphore is an integer associated to a given process. Its purpose is to memorize all the signals sent to that process but not yet received by it.

Two primitives are associated to a private semaphore :

wait : sem : = sem - 1 ;
if sem < 0 then process state(running) : = blocked.

signal (p) : sem : = sem + 1 ;
if sem = 0 then process state(p) : = ready

The following rules must be respected to use a private semaphore :

1. A private semaphore cannot be initialized to a negative value.
2. The wait operation can be executed only by the owner of the semaphore. Consequently, a private semaphore cannot take a negative value other than -1.

Remark :

To initialize a private semaphore to a positive value (say N) means that the owner process will have to execute N wait operations before signals sent by other processes can be taken into consideration.

### 1.3.5. Synchronization by public semaphores (BH2, Cr, D)

Like a private semaphore, a public semaphore is an integer which cannot be initialized to a negative value. The difference with a private semaphore is that the wait operation (we call it P) can be executed by every process and not only by a particular process. For that reason we associate a queue of process to such a semaphore.

The primitives associated to a public semaphore are called P and V and are defined as follows ( s is the name of the semaphore) :

P(s) : s = s - 1
if s < 0 then block running process and put it in the semaphore queue.

V(s) : s = s + 1
if s ⩽ 0 then get a process from the queue and make it ready.

Notes

1. By definition of V, only one process at a time will leave the semaphore queue if it is not empty.

2. A positive value of the semaphore indicates the number of signals sent but not yet received.

3. If a semaphore is negative, its absolute value indicates the number of processes waiting in the queue.

Remark

We make no assumption about the order in which processes enter or leave the waiting queue.
In our model (see next chapter) we will use a FIFO scheduling.

## 1.3.6. Comparison of synchronizing tools

| Characteristics | Wakeup-waiting switches | Events | Private semaphores | Public semaphores |
|---|---|---|---|---|
| - Only one process can wait for a given signal. | X | | X | |
| - Every process can wait for a given signal. | | X | | X |
| - Blocked processes must enter a waiting queue associated to the signal type. | | X | | X |
| - All occurrences of a signal are memorized. | | | X | X |
| - Only one occurrence of a signal can be memorized before it is taken into consideration. | X | X | | |
| - The identity of the receiver must be known by the sending process (the signal is sent directly to a given process). | X | | X | |
| - The receiver does not know the identity of the sending process. (1) | X | X | X | X |
| - A wake-up signal awakes all processes waiting for it. | | X | | |
| - A wake-up signal awakes one process only | X | | X | X |
| - A wake-up signal is not necessarily lost after activation of processes waiting for it. (2) | | X | | |
| - A process can block itself even if the signal has occurred. (3) | | X | | |
| - Processes waiting in a queue associated to a signal type, may be scheduled at the arrival of the signal. | | | | X |

(1) The identity of the sending processes can be known by sending messages to the receiver. (see 1.6.)

(2) see 1.3.3. : procedure awake(event)

(3) see 1.3.3. : procedure block(event)

## 1.4. Mutual exclusion

### 1.4.1. Critical regions (BH2)

We need a mechanism which prevents processes to manipulate simultaneously a shared object. Such a mechanism is called Mutual Exclusion Mechanism.
We define a critical region as a set of operations which cannot be executed at the same time ; we will say that a process which wants to manipulate a shared object can do it only inside a critical region associated to that object.

We make the following assumptions about critical regions :

1. Only one process at a time can be inside a critical region.
2. A process which is inside a critical region must leave it after a finite time.
3. A process which was not allowed to enter a critical region must be able to do it after a finite time.

NOTES

1. Different critical regions can be executed simultaneously by different processes.
2. A process can enter nested critical regions.
3. The last remark can lead to a deadlock situation (see 1.5.)
   Example : see next page.

EXAMPLE

| Process P1 | Process P2 |
|---|---|
| enter region A | enter region B |
| . . . | . . . |
| enter region B | enter region A |
| . . . | . . . |
| leave region B | leave region A |
| . . . | . . . |
| leave region A | leave region B |

If process P1 enters region A at the same time as process P2 enters region B, there will be a deadlock situation when P1 will try to enter region B and P2 region A.

Remark

Deadlocks of nested critical regions can be prevented by a hierarchal ordering of critical regions (see 1.5.)

### 1.4.2. Implementation of critical regions

We suppose a multiprocessor environment.

#### 1.4.2.1. Test-And-Set (TAS) instruction

To each critical region we can associate a boolean variable. When the variable has the value 0, it means that no process is inside the critical region, so that the region can be entered by the next requesting process.

When a process wants to enter a critical region, it tests the variable associated to it (say X) and enters the critical region only if the variable has the value 0, but before entering the region, it must lock it by giving the

value 1 to the variable. This is done by the TAS instruction which can be defined as follows :

```
TAS(X)  :  if  X = 0 then  begin
                             X : = 1 ;
                             skip next instruction
                           end
```

Thus, before entering a critical region, a process must execute the following sequence :

```
loop    :  TAS(X) ;
           goto loop ;
region  :  . . .
```

Before leaving a critical region, a process must open it again by executing the statement X : = 0

Remarks

1. The boolean variable associated to a critical region must be initialized to 0.
2. There is a danger of deadlock is we allow a process to be interrupted when it is inside a critical region. For example, let us consider a process P1 which is inside critical region A. At the same time an interrupt signal forces a processor to execute an interrupt routine which has to enter the critical region A. If the processor is the same as the one used by P1, the critical region will never be freed and the interrupt routine will be in a deadlock situation with P1.

   To avoid such a situation, we could, for example, mask all interrupts and define the mutual exclusion mechanism as follows :

```
        disable interrupts
loop    : TAS(X)
          goto loop
region  : . . .
          X : = 0
        enable interrupts
```

3. This mechanism has the following disadvantages :
   - it forces a process to execute a waiting loop.
   - it disables the interrupts during the time of this loop.

Note

The waiting-loop duration will depend of the time the process which is inside the critical region will stay in that region. If that time is long enough, it may be interesting to force the requesting processes to enter a queue associated to the critical region. The locks and mutual exclusion semaphore mechanisms will do that.

Remark

Instead of disabling the interrupts when a process is inside a critical region, we could permit them, but force the interrupt routine to enter an "interrupt waiting queue" associated to the critical region, when it tries to enter that region, and give the control of the processor back to the interrupted process. On the other side, that process should liberate the waiting interrupt routine before it leaves the critical region, in such a way that the routine can continue its execution inside the critical region.

For example, let us consider a process X executing instructions inside a critical region A.

If interrupts are not disabled, an interrupt signal could give control of the processor to an interrupt routine Y which, at a certain point of its execution, has to enter the critical region A.
At that moment, the interrupt routine could execute the following sequence of instructions :

```
                . . .
                TAS(A) ;
                goto wait ;
                goto region ;
wait        : enter interrupt routine into an interrupt
              waiting queue associated to the critical
              region A and give control of the proces-
              sor to the interrupted process.
region      : . . .
                 executes instructions inside critical
                 region A.
              A : = 0
```

Before leaving the critical region A, process X should execute the following statement :

```
if interrupt waiting queue is empty
   then A : = 0
   else give control of the processor to the inter-
        rupt routine waiting in the queue.
```

### 1.4.2.2. Locks

A lock is a boolean variable to which is associated a queue of processes. Its purpose is to suppress the waiting-loop of processes which were not allowed to enter a critical region, by putting them into a waiting queue.

The operations on a lock can be defined as follows :

```
lock (X)   : if  X = 0  then  X : = 1
                        else block running process
                             into the queue.

unlock (X): if  queue = empty
            then  X : = 0
            else get a process from the queue
                 and make it ready.
```

Notes

1. A lock itself must be protected against simultaneous manipulation by more than one process at a time. To do so, we must associate to it a boolean variable that must be tested by a TAS instruction.

2. A lock will be opened (take the value 0) only when its processes queue is empty.

### 1.4.2.3. Mutual Exclusion Semaphores

A mutual exclusion semaphore, mutex, is a public semaphore associated to a critical region. As for the locks, its purpose is to block into a waiting queue all processes which could not enter a critical region.

The following rules must be respected to use a mutual exclusion semaphore :

1. It must be initialized to 1.
2. When a process wants to enter a critical region, it must execute P (mutex)
3. When a process wants to leave a critical region, it must execute V (mutex).

Consequently, such a semaphore has the following properties :

a) It cannot take a positive value $> 1$.
b) It has the value 1 when no process is inside the critical region.
c) It has the value 0 when one process is inside the critical region and the semaphore queue is empty.
d) It has a negative value when processes are waiting in the queue, to enter the critical region.

## 1.5. Deadlocks

### 1.5.1. Definition (BH2, Cr, M&D)

A deadlock is a situation in which two or more processes are waiting indefinitely for resources held by the others.

A process is expected to make a request for a resource before it uses them. The request operation delays the process until the resource is available. When the process has no more need of the resource, it must release it by executing a release operation.

Example of deadlock

Let us consider 2 processes A and B which are sharing a printer and a reader by means of the request and release operations :

| Process A. | Process B. |
|---|---|
| A1 : request printer | B1 : request reader |
| A2 : request reader | B2 : request printer |
| A3 : release printer | B3 : release printer |
| A4 : release reader | B4 : release reader |

A deadlock situation will occur if the request and release operations are executed in the following order :

A1 B1 A2 B2 X

In A2, process A must block because the reader has been required by B at B1. In B2, process B must block because the printer has been acquired by A. So, in X, the two processes are blocked, each waiting for a resource that can be released only by the other.

### 1.5.2. Deadlock Prevention

It can be shown that the following conditions are necessary for the occurrence of a deadlock situation (BH2) :

1. Mutual Exclusion : A resource can only be acquired by one process at a time.
2. Non-preemptive scheduling : A resource can only be released by the process which has acquired it.
3. Partial allocation : A process can acquire its resources piecemeal.
4. Circular waiting : Processes have acquired part of their resources and enter a state in which they wait indefinitely to acquire each other's resources.

Deadlocks can be prevented by insuring that one or more of the necessary conditions will never hold.

We will define a method that prevents circular waiting by imposing a sequential ordering of requests (Hierar-Chal Resource Allocation).

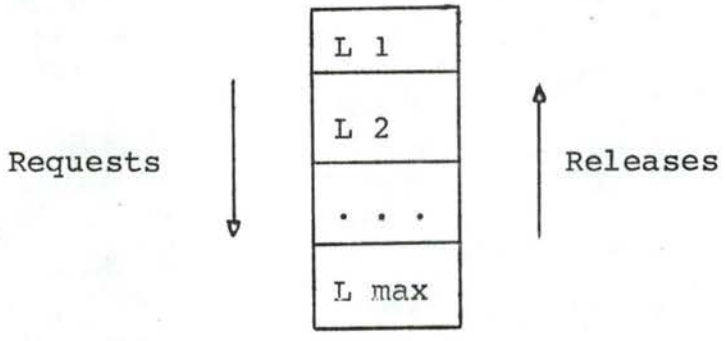
### 1.5.3. Hierarchal Resource Allocation (BH2, Cr)

In this method resources are grouped in hierarchal classes of levels L1, L2, ... Lmax.

Deadlocks will be prevented by respecting the following rules :

1. A process must acquire all resources it needs from a class, by a single request.
2. When a process has acquired resources from a class Lj, it can only acquire resources from a class of superior level Lk ($k > j$).
3. Resources acquired at a level Lk must be released before the resources acquired at a lower level Lj ($j < k$).
4. When a process has released all resources acquired from a class, it can request other resources of the same class.
5. Resources must be released after a finite time.

The hierarchal resource allocation may be schematized as follows :



Example (M&D)

We assign a unique number to all resources of the system :

reader = 1
printer = 2
punch = 3
tape = 4
disk = 5

All requests must be in ascending order and releases in descending order :

| | | | |
|---|---|---|---|
| correct sequence | : | request reader | (1) |
| | | request punch | (3) |
| | | request tape | (4) |
| | | release tape | (4) |
| | | release punch | (3) |
| | | release reader | (1) |
| illegal sequence | : | request reader | (1) |
| | | request tape | (4) |
| | | request punch | (3) |

One disadvantage of this method is that the standard sequence does not necessarily correspond to the actual ordering of resources utilization.
For example, a process may want to use the tape before the printer. Nevertheless, the printer must be requested before the tape.

## 1.6. Process communication (Ha, BH2, Cr)

Synchronization mechanisms allow processes to exchange signals but not messages nor data.
If we define a buffer as an area of memory reserved to contain messages, the exchange of messages between processes can be done by using a buffer, shared by the sending and the receiving processes.

In this chapter, we consider a communication buffer structured as a circular linked list of n message frames, where $n \geq 2$ (Ha). We make no assumption about the number of processes which deposit messages in the buffer, nor about the number of processes which remove messages from that buffer. A process which deposits a message is called a "producer" and a process which removes a message is called a "consumer".

Two pointers are associated to the buffer : "front" points to the first empty message frame when no message is being placed, and "rear" points to the first full frame when no message is being removed.

The following remarks must be taken into consideration prior to defining the deposit and remove procedures :

1° Buffer overflow : a producer cannot deposit a message if the buffer is full. We define a semaphore "frame" that we initialize to n, the buffer capacity (Number of message frames). Buffer overflow will be avoided if the sending process executes the P(frame) operation prior to depositing its message.

2° Buffer underflow : a consumer cannot remove a message from an empty buffer. We define a semaphore "message" that we initialize to 0. Buffer underflow will not occurred if the receiving process executes the P(message) operation prior to removing a message.

3° Several producers must not be able to deposit a message simultaneously. Therefore the deposit operation must be programmed as a critical region that allows only one sender at a time to place a message. A mutual exclusion semaphore "mutexprod" will assure that protection.

4° Several consumers must not be allowed to accept a message simultaneously. This will be achieved by a mutual exclusion semaphore "mutexcons" that protects the remove operation as a critical region.

The scheme producers-consumers may be defined as follows : (in that scheme, "succ" is a successor function such that, if x is a pointer to a message frame of the buffer, succ(x) points to the next frame in the buffer).

```
Producers                                   Consumers

        semaphore frame = n, message = 0,
                  mutexprod = 1, mutexcons = 1 ;
        pointer   rear : = front : = first message frame
                                        pointer ;

procedure deposit(d) ;                  procedure accept(r) ;
begin                                   begin
   P(frame) ;                              P(message) ;
   P(mutexprod) ;                          P(mutexcons) ;
   buffer(front) : = d ;                r : = buffer (rear) ;
   front : = succ(front) ;              rear : = succ(rear) ;
   V(mutexprod) ;                       V(mutexcons) ;
   V(message)                           V(frame)
end                                     end
```

If that scheme is respected, it can be proved that (Ha) :

1° a producer and a receiver cannot access a same message frame simultaneously.

2° there is no danger of deadlock between producers and consumers.

Special case : n = 1

If the buffer is reduced to one message frame, the pointers "front" and "rear" are no more necessary.

It can also be shown (Ha) that :

1° the deposit and accept operations cannot be executed simultaneously.

2° the semaphores "mutexprod" and "mutexcons" are no more necessary.

Therefore, the scheme Producers-Consumers can be reduced to the following :

```
Producers                          Consumers

            semaphore frame = 1, message = 0 ;

procedure deposit(d) ;             procedure accept(r) ;
begin                              begin
   P(frame) ;                         P(message) ;
   buffer : = d ;                     r := buffer ;
   V(message)                         V(frame)
end                                end
```

Remarks

Communication by means of a cyclical message buffer is convenient if the consumers are all equivalents (Cr), that is if any message can be received by any consumer. Otherwise, the consumer identity should be defined in the message and a sort should be done at the accept operation.
A solution to avoid such a sort is to assign a different buffer to each class of equivalent consumer processes.

# CHAPTER 2 : A Real-Time Monitor for Multiple Processor Microcomputer Systems

## 2.1. Introduction

The purpose of this chapter is to define a real-time monitor for multiple processor microcomputer systems.
The monitor has been conceived in order to be implemented on a model of configuration offered by the best known of the microprocessor constructors.
Such a configuration includes a common bus that allows each microprocessor to access a common memory and to share peripherals. On the other hand, a local bus is assigned to every processor so that it can access private memories and I/O.

## 2.2. Limits

We made the following assumptions prior to defining the monitor :

1°) The number of processes is fixed. There is no dynamic creation nor deletion of processes.

2°) A process can execute its programs on one processor only.

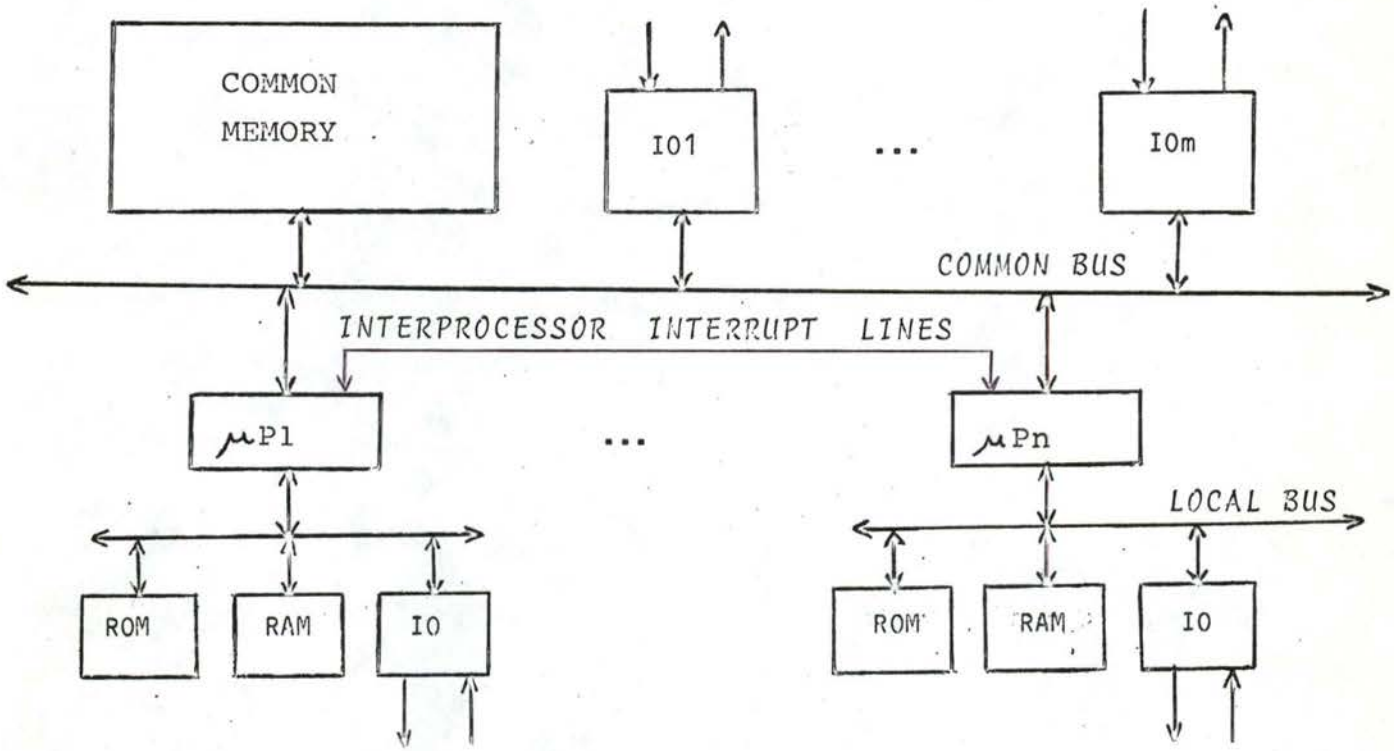
3°) The common memory is essentially used for interprocess communication and synchronization.

4°) The programs are fixed, once for all, in private ROM memories. This is done to limit the common bus accesses.

5°) The problems of objects protection and of deadlock prevention are essentially resolved at design and compilation time.

## 2.3. Hardware Configuration

We assume that the processes will run on a configuration that can be schematized as follows :




In that configuration :

- Each microprocessor ($\mu$P1,...,$\mu$ Pn) has access (via a bus arbitration logic, not represented here), to a common system bus to which are connected a common memory and IO controllers (or other logics, like high speed mathematical functions modules, for example).
- On the other side, each microprocessor has access, via a local bus, to private memories (ROM and RAM) and IO.
- Interprocessor interrupt lines allow the processors to send interrupt signals, from one to each other.
- Different types of microprocessors may be connected to the system.

## 2.4. Memory organization

### 2.4.1. Common Memory

The main elements that are under control of the monitor procedures are the following :

- processes
- processors
- semaphores
- events
- peripherals
- a common mailbox.

These elements are represented by records of given type :

- A process is represented by a record of type "process".
- A processor by a record of type "processor".
- etc.

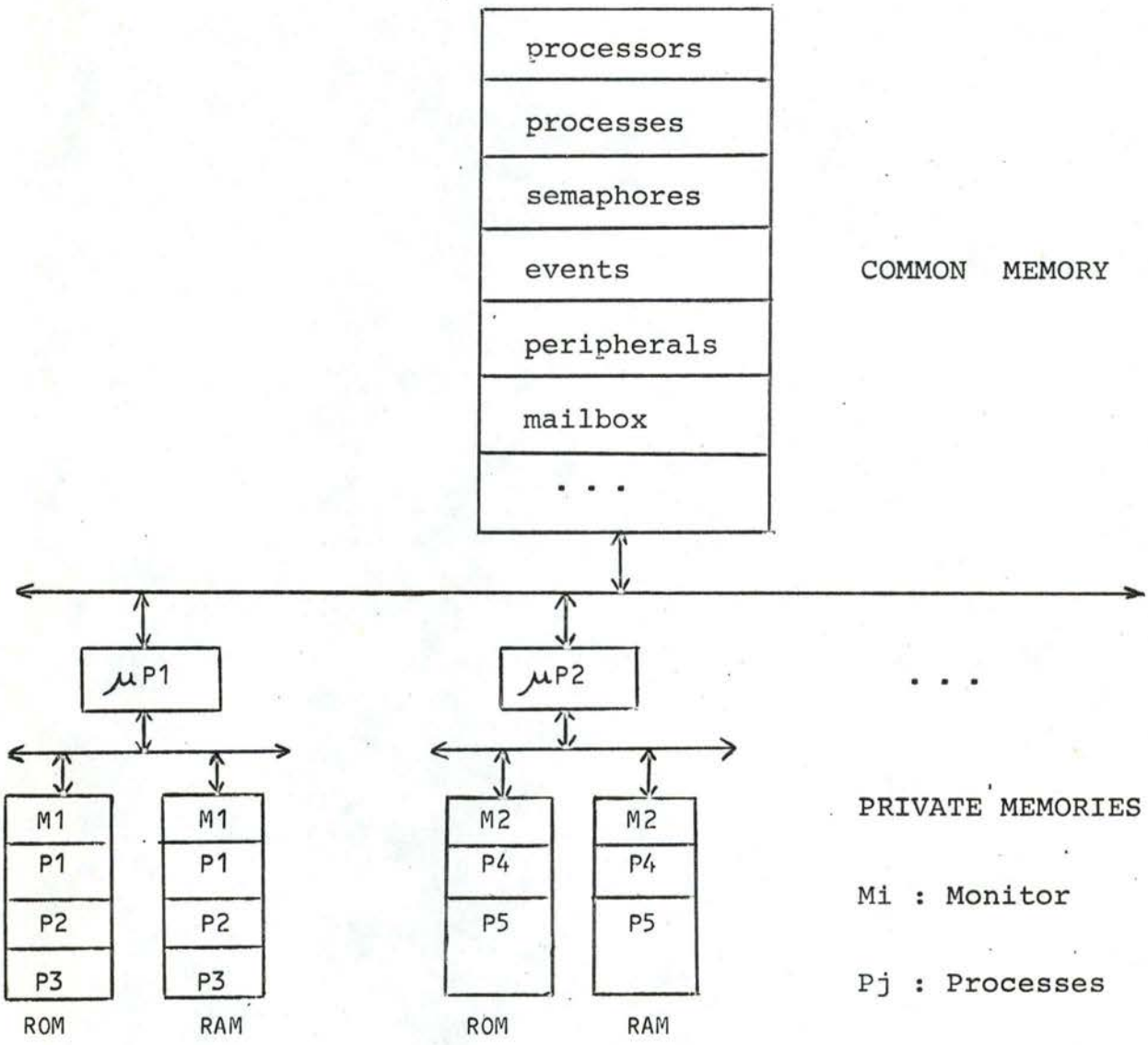
We assume that all these records are located in the common memory.

### 2.4.2. Private memories

A private memory is accessible to one processor only. It contains the monitor procedures and the programs and private data of processes runnable on that processor. Thus, there is a private monitor for each processor. Although each of them must respect the global rules defined by the system designer, the procedures may be adapted, when necessary, to satisfy the requirements of processes that must execute them, with the greatest flexibility.

### 2.4.3. Scheme

Therefore, the memory organization can be schematized as follows :




In that scheme, we have sectionned the private memories into ROM and RAM. The ROM memories contain the monitor or processes procedures or constants. The RAM memories contain the monitor or processes private variables or working areas.

NOTE The common memory could also be divided in ROM and RAM. We assume that the represented objects are located in the RAM part of that memory.

## 2.5. User processes and Home processes

We have defined a process as an entity which has to execute a program on a processor. We shall now extend that definition by saying that every program or procedure must be executed under control of a process, but we shall separate the processes in two groups : the "user processes" and the "home processes".

A user process is materialized in the system by a record of type "process" (see 2.10) and must respect all the coordination rules defined in the monitor procedures. In particular, when they do not use a processor, they must wait in a waiting queue of processes such as a ready queue or a semaphore or event queue.

A home process is not materialized by a record of type "process", so that it has not to follow the same set of rules that the user processes. It can be identify by a set of entry points to procedures, so that control of the processor can be given to it directly without passing to a control procedure which has to take it from a waiting queue of ready processes. Examples of such processes are processor controllers or processes awakened by an interrupt signal to execute an interrupt routine.

A home process cannot execute the synchronization procedures to block itself, but it will use them to awake user processes, waiting in a queue of processes.

One advantage of such a process is that it can execute its activity with more flexibility than the user processes. In particular this flexibility can be used to respond to an external signal in the shortest period of time.

## 2.6. Process States and State Diagram

The following rules concern the user processes only.

### 2.6.1. Process States

A process can take four states :

- running
- ready
- blocked
- stopped.

#### 2.6.1.1. Running state

A process is running when a processor is executing its program.

An information contained in the processor record indicates to the monitor which is the process currently running on that processor.

A running process may be interrupted to give temporarily the processor control to a home process. In such a case, it does not lose its state but is still considering as running, unless it is preempted or stopped by the home process which obtained control of the processor.

When there is no user process running on a processor, control of it is given to a home process named "The Processor Controller" There is such a process for each processor. Its function is to examine the ready queue assigned to the processor it controls. When it finds that the queue is not empty, it will select the most priority process waiting in the queue and make it running.

2.6.1.2. Ready State

A process is ready when it is waiting, in a ready queue of processes, for its turn to get control of the processor.

There is a separate ready queue by processor.

A process is made ready when it is awakened by a synchronization signal (semaphore or event) or by a start operation, or - if it is running - when it is preempted by a home process to give control of the processor to a more priority process.

2.6.1.3. Blocked State

A process is blocked when it is waiting, in a semaphore or event queue of processes, for a synchronization signal that will make it ready.

A process can only block itself, by executing a P(s) or a wait(event) operation.

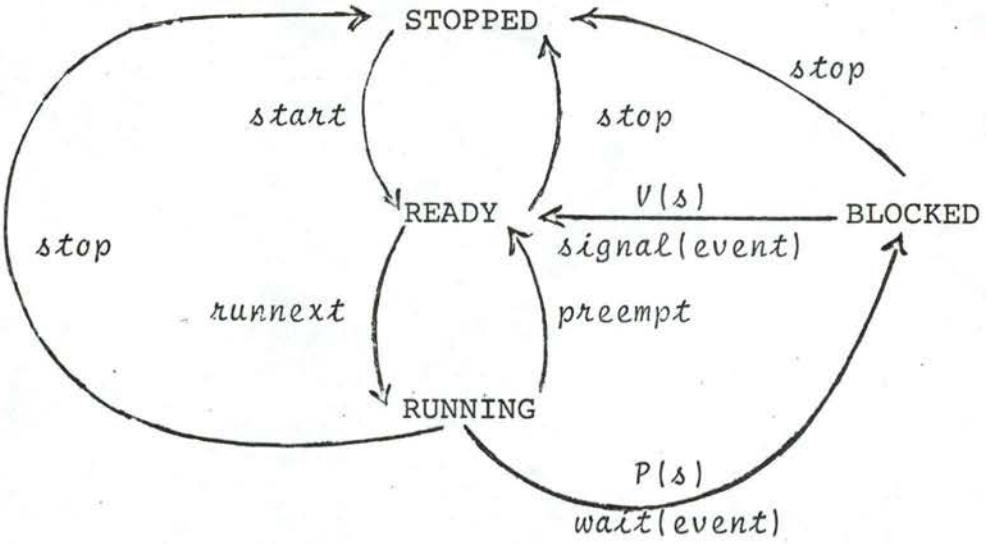
2.6.1.4. Stopped State

A process in the stopped state is not runnable until it is made ready by a start operation.

A process may be stopped whatever its current state is, but it cannot be stopped if it is inside a critical region. In such a case the stopping operation is delayed until the process leaves the critical region.

2.6.2 State Diagram

The process State Diagram may be represented as follows :




This diagram may be interpreted as follows :

- The states (stopped, ready, running and blocked) are indicated in capital letters.
- The operations which allow a process to pass from one state to another are indicated in small letters.

The following rules appear from the diagram :

- A process may be stopped, whatever its current state is (ready, running or blocked).
- A stopped process, when restarted, is made ready by the start operation.
- A ready process is made running by the "runnext" operation (executed by the "Processor Controller").
- A running process may be preempted.
- A running process may block itself, by executing a P(s) or wait(event) operation.
- A ready process cannot be blocked.

- A blocked process cannot pass directly into the running state. It must pass through the ready state, first.
- A blocked process can be awakened by a V(s) or signal (event) operations (these can be executed by home or user processes).

## 2.7. Short overview of Pascal

The procedures and data structures described in that chapter are written in Pascal. That language is used as a *conceptual* language only.

A short overview of the main elements of that language is given below. (A most complete overview can be found in (BH3)).

### 2.7.1. Program structure

A Pascal program consists of declarations of

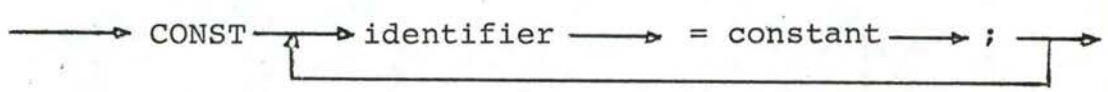
constants
data types
variables
routines

and a sequence of statements that operate on these objects.

### 2.7.2. Constants

A constant represents a value that can be used as an operand in an expression.

Const definition :

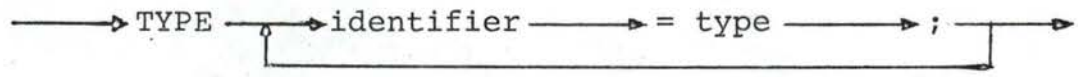



example : const pagelength=512;firstline=2;

## 2.7.3. Types

A data type defines a set of values which may be assumed by a variable or an expression.

type definition :




A type definition introduces an identifier as the name of a data type. A data type cannot refer to its own type identifier.

### 2.7.3.1. Simple data types

Enumeration type

An enumeration type consists of a finite, ordered set of values.
Examples :

```
type integer =(-32768,...0,1,...32767)

type boolean = (false,true)

type char = (nul,...,&,...'0','1',...'a','b',...del)

type iodevice = (typedevice,printdevice,tapedevice).
```

This type definition introduces a new data type called "iodevice". Its values are called "typedevice", "printdevice" and "tapedevice".

Other examples :

```
type iooperation = (input,output,move,control)

type ioresult = (complete,intervention,endfile,...)
```

### 2.7.3.2. Structured data types

Arrays and records are data structures composed of simpler types. They can be operated upon either as a whole or component by component.

array

An array is a data structure with a fixed number of components of the same type.
For example, a text line can be defined as an array of character :

```
type line = array(.1..132.)of char
```

The declaration

```
var text:line;
```

introduces a line variable "text".

record

A record is a data structure with a fixed number of components that may be of different types.
For example, to output a line on a printer one uses a record that defines the input/output operation and its result :

```
type ioparam = record
                 operation:iooperation;
                 status:ioresult;
                 arg:integer
                 end
```

This data type is called an "ioparam". It contains three fields named "operation", "status", and "arg". These fields are of types "iooperation", "ioresult", and "integer" defined earlier.

A record field is selected by means of its identifier.
For example,let us suppose that "param" is a variable of type "ioparam". The fiels "operation" and "status" may be selected as follows :

```
param.operation
param.status
```

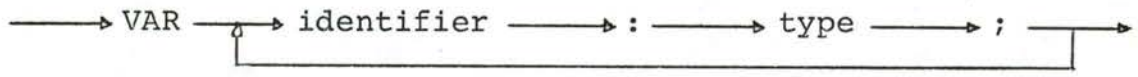
Instead of repeatedly qualifying record fields with the same record identifier one can do it once by means of a "with" statement :

```
with param do
begin
  operation:=output;
  repeat io(text,param,printdevice)
  until status=complete;
end
```

### 2.7.4. Variables

A variable is a named abstract store location that can assume values of a single type. The basic operations on a variable are assignments of a new value to it and a reference to its current value.

var definition




Example :

```
var param:ioparam;i:integer;c:char;
```

### 2.7.5. Pointer types

Most of the objects defined in the monitor procedures will be referenced by pointers.

A pointer type P consists of an unbounded set of values pointing to elements of a given type T. P is then said to be "bound" to T.
The value "nil" is always an element of P and points to no element at all.

A pointer type will be identified by the symbol "&" followed by the type name of the record it points to and that record

will be referenced by the pointer name followed by the same symbol "&".

For example, if p is a pointer variable bound to a type T by the declaration

```
var p:&T
```

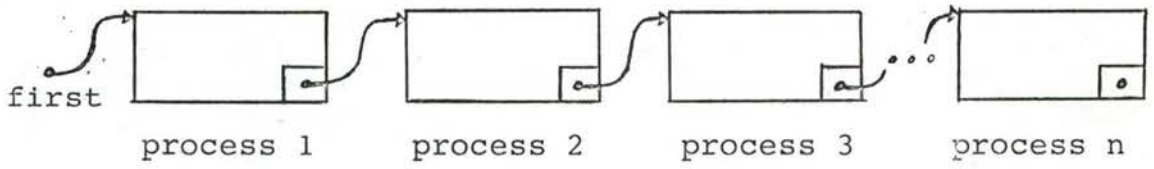
then p is a reference to a variable of type T, and p& denotes that variable.

Example :

The process records are chained together by variables of type "processpointer" :

```
type processpointer=&process;
type process=record
                ...
                next:processpointer;
                ...
             end
```

"next" points to the next process record in the list :




A variable of type "processpointer", called "first" points to the first element of the list. The link of the last process is "nil".
The first element in the list is denoted by : first&, the second by (first&.next)& etc...

## 2.8. Pascal_extension

We will use the word "shared" to indicate that an object is located in the common memory :

Example :

```
type process = shared record ... end
```

This process type definition denotes that all the records of type process are located in the common memory.

## 2.9. Mutual Exclusion

### 2.9.1. Locking the common bus

We assume that two hardware commands allow a processor to obtain exclusive control of the common system bus as long as it is needed. These commands are defined as follows :

lockcommon
unlockcommon

"lockcommon" locks the common bus in such a way that the bus is accessible only to the processor that executed that command and until it issues an "unlockcommon" instruction.

### 2.9.2. Test-And-Set Instruction

The "lockcommon" and "unlockcommon" allow us to implement the "TAS"instruction defined in chapter 1.

The TAS procedure locks and unlocks continually the common bus until the boolean variable that it must test has the value "false". In that case, the value "true" is given to the variable prior to unlocking the bus.

ALGORITHM 1

```
procedure tas(x:boolean);
begin
  repeat
     lockcommon;
     if x then unlockcommon
  until not x;
  x:=1;
  unlockcommon
end
```

To reduce the number of memory locks we could modify the above algorithm in such a way that the process will lock the bus after it found that the boolean variable has the value "false". An inconvenience of such a solution is that, if several processor with different speeds are executing a TAS instruction on the same variable, the lower speed processors would test the variable less frequently that the others, resulting in an "unfair" advantage for the higher speed processors :

Algorithm 2

```
procedure tas(x:boolean);
begin
  repeat
     if not x then
     begin
       lockcommon;
       if not x then begin
                        x:=1;
                        unlockcommon;
                        return
                        'exit from the procedure'
                     end
                  else unlockcommon
     end
  forever 'try again'
end
```

### 2.9.3.Mutual Exclusion Semaphores

The TAS instruction allows us to protect as many critical regions as we wish. In particular, it will be used to implement the semaphores. As we saw in Chapter 1, semaphores can also be used to assure the mutual exclusion.

### 2.9.4. Deadlock Prevention

To prevent deadlock we group the critical regions and the peripherals into hierarchal classes.

For the monitor procedures, the classes have been defined as follows :

1 : Semaphores, events
2 : processors
3 : processes

For the peripherals, the classes will be defined at the application design time.

The monitor will not verify if the requests to peripherals or critical regions are respecting the hierarchal order that prevent deadlock.
This verification must be done at compilation time.

## 2.10. User Processes

### Process description

A User Process is represented in the common memory by a shared record of type "process" :

```
type process =
   shared  record
           protect : boolean ;
           name : character ;
           state : (ready, running,
                    blocked, stopped) ;
           priority : integer ;
           processor : &processor ;
           stopwaiting : boolean ;
           regions : integer ;
           iocount : integer ;
           next : &process ;
           globalnext : &process ;
           waitingqueue : synchroqueue
           requestqueue : &peripheral ;
           mutexmail ; &semaphore ;
           firstmessage : &frame ;
           fullmail : &semaphore ;
           emptymail : &semaphore ;
           stopper : &semaphore
           context : processorregisters
           nextrequester : &process
      end
```

The fields are defined as follows :

protect : This field contains a boolean value. It is used to protect the process record (by a TAS instruction) inside a critical region.

name : A process is normally referenced by the address of its record. This field could be used to reference a process by name rather than by its address.

state : This field contains the process state value. A process can take four states : ready, running, blocked or stopped.

priority : This is an integer which determines the priority of the process in relation to the others. A small value in this field gives a high priority to the process. It is used by the processor controller to select the most priority process from the ready queue associated to the processor it controls.

processor : A process can execute on one processor only. This field points to the processor record associated to the processor on which the process can run. It is used by the V and signal primitives to find the address of the ready list into which an awakened process must be inserted.

stopwaiting : This field is a boolean used to indicate that the process is waiting to be stopped. It has the value "true" when a stop operation on that process could not complete because the process was inside a critical region or in the ready state. In the last case it will be stopped definitively by the processor controller. (see 2.13.)

regions: This field indicates the number of nested critical regions entered by the process. It is incremented by one each time a process has to enter a critical region without disabling all interrupts, and decremented by one each time it leaves such a region. There are primitives which allow a process to enter such critical region. (see interregion and leaveregion procedures). The process cannot be stopped if that field contains a nonzero value.

iocount : This field is an integer which indicates the number of I/O operations initiated by the process but not yet completed.

next : All processes waiting on a same semaphore or event are chained together in a linear list. The head pointer of the list is situated in the semaphore or event record. This field links the process to the next process in the list. Its value is "nil" if it is the last process in the list.

globalnext : All processes records are chained independently together by this link, in a linear list which can be used for processes management.

waitingqueue : This is a pointer to the semaphore or event record which contains the head pointer of the process queue in which the process is waiting, when it is in the blocked state. Because "waitingqueue" can point to records of different types (event or semaphore), it must be defined (this is a Pascal constraint (BH3))as a record that contains either a semaphore pointer or an event pointer. This record must include a "tag field" that defines which of the two "variants" (event or semaphore pointer) is being represented by the other record field (BH3). It may be defined as follows :

```
type  synchroqueue =
      record
          case tag:(semtype,eventtype) of
             semtype:(sema:&semaphore);
             eventtype:(evt:&event)
      end
```

This notation may be interpreted as follows:
If the tag field has the value "semtype", then the rest of the record is a semaphore pointer named "sema".
On the other hand, if the tag value is "eventtype",then the rest of the record is an event pointer named "evt".
Although we do not intend to use a Pascal compiler to implement the kernel, we will use this variant record to recognize the type of record that is pointed to by the "waitingqueue" field.

requestqueue : This field is a head pointer to the list of peripherals acquired by the process by a peripheral request procedure call. When a new peripheral has been acquired by the process, its record is inserted at the beginning of that list. A release operation will remove the first peripheral from that list (LIFO). This organization assumes that the peripheral requests and releases follow the rules of hierarchal allocation of resources.

mutexmail : This mutual exclusion semaphore is used to protect the process message chain, pointed by the "firstmessage" field, against its manipulation by more than one process at the same time. This semaphore is initialized to 1. (see interprocess communication).

firstmessage : All the messages sent to the process but not yet received by it, are chained together in a linear list pointed to by this head pointer. This list is called the "message queue" of the process.

fullmail : This semaphore is initialized to 0. Whenever a message is sent to the process, a V operation is performed on the semaphore by the sending process, so that the semaphore value indicates the number of messages sent to the process but not yet received by it. Whenever the receiver process attempts to read a message from its message queue, it performs a P operation on that semaphore, so that it will block itself if the queue is empty.

emptymail : This semaphore is initialized to the maximum number of messages that can be linked together in the process message list. Whenever a process attempts to send a message to that receiver process, it performs a P operation on that semaphore, so that it will block itself when the semaphore is negative, that is if the message list of the receiver process is "full".
On the other hand, the receiver process will perform a V operation on that semaphore, each time it has received a message from the list.

stopper : This semaphore is used by the process whenever it attempts to stop a process which is not runnable on the same processor. In such a case the stopping process sends an interrupt signal to the other processor and blocks itself on that semaphore. It will be awakened by a V operation executed by a home process on the other processor after reception of the interrupt signal.
The semaphore is initialized to 0.

context : This field is a save area used to save the context of the process when it has to leave the processor on which it is running. We define a process context as the set of values contained in the external registers that can be accessed by the process (processor registers, masks, etc...). As the context may vary from one processor to the other, this field must be defined, like the "waitingqueue" field, as a variant record of type "processoregisters" :

```
type  processorregisters =
      record
        case tag :(type1,....,typen) of
          type1:(reg1:record...end)
          ...
          typen:(regn:record...end)
      end
```

type1,...,typen define the different processor types that are connected to the system. To each type is associated a record that defines the registers that can be accessed by a process running on a processor of that type.

nextrequester : All processes that have requested a peripheral but could not be satisfied because the peripheral had been reserved by another process, are linked together by this field, in a list pointed by a head pointer contained in the peripheral record.

## 2.11. Queues of processes

### 2.11.1. Definition

A queue of processes is a linear list of process records chained together by pointers contained in the process records themselves. The first element of the list is pointed to by a pointer field called the "head pointer".

### 2.11.2. Operations on processes queues

We will use the following functions or procedures to operate on processes queues :

#### 2.11.2.1. function empty(head:&process);

This function returns the value "true" if the head pointer contains the value "nil" or "false" if it does not.

ALGORITHM 3

```
function empty(head:&process);
begin
   if head : = nil then empty(head):=true
      else empty(head):=false
end
```

#### 2.11.2.2. procedure put(p:&process,head:&process);

This procedure puts the process p at the end of the process queue pointed by head, so that the process becomes the last element of the list.

ALGORITHM 4

```
procedure put(p:&process,head:&process);
var q:&process ;
begin
   if empty(head) then head:=p
      else
          begin
            q:=head;
            while q&.next >< nil do
            begin
              q:=q&.next
            end;
            q&.next:=p;
          end
      with p& do
      next:=nil;
end
```

2.11.2.3. procedure get(p:&process,head:&process)

This procedure gets the first element of a process queue. It is used, with the "put(p:process,head:&process)" procedure, to schedule a FIFO queue of processes.

ALGORITHM 5

```
procedure get(p:&process,head:&process);
begin
   p:=head;
   head:=p&.next
end
```

2.11.2.4. procedure enter(p: &process,head:&process)

This procedure inserts a process on the top of a process queue, so that the new process becomes the first element of the list.

ALGORITHM 6

```
procedure enter(p:&process,head:&process);
begin
   p&.next:=head
   head:=p

end
```

2.11.2.5. procedure select(p:&process,head:&process)

This procedure removes the most priority process from a non empty queue of processes. If the queue contains processes with the same priority, the procedure will remove the one which is nearest the end of the queue, so that, if processes entered the queue via the "enter(p:&process, head:&process)" procedure, the selected process will be the first which arrived into the queue. (algorithm 7)

ALGORITHM 7

```
procedure select(p:&process,head:&process);
var x:integer;plast,q:&process;
begin
   p:=head;plast:=loc(head);q:=p;
   x:=p&.priority;
   while q&.next >< nil do
   begin
      if (q&.next)&.priority =< x then
         begin
           x:=(q&.next)&.priority
           plast:=q;
           p:=q&.next
         end ;
      q:=q&.next
   end ;
   if p=head then head:=p&.next
             else plast&.next:=p&.next
end
```

## 2.12. Processor

### 2.12.1. Processor description

A processor is represented in the common memory by a shared record of type "processor" :

```
type  processor = shared  record
                    running:&process;
                    readyprotect:boolean;
                    readyqueue:&process;
                  end
```

The fields are defined as follows :

running : This field contains the process record address of the user process that is running on the processor. It has the value "idle" if there is no such a process.

readyprotect : This field contains a boolean value which is used to protect the ready queue of that processor (by a TAS instruction) against its manipulation by more than one process at the same time.

readyqueue : This is the head pointer of the ready list associated to the processor. If the list is empty, the pointer has the value "nil".

NOTE :

We assume that the procedure defined in these pages are executed on a processor which record is pointed by Q :

```
var Q:&processor
```

We shall call that processor the processor "Q".

## 2.12.2. Processor Utility Routines

The procedures defined in the next pages will call the following routines :

- disableinterrupts : This routine disables the interrupts on the processor on which the routine is executed.

- enableinterrupts : This routine enables the interrupts on that processor.

- saveregisters : This routine saves the processor registers in a save area defined in the private memory of that processor.

- restoreregisters : This routine restores the contents of the save area into the processor registers.

- savecontext : This routine transfers the contents of the processor save area into the running process record.

- restorecontext(p:&process) : This routine transfers the content of process p from the record defining that process into the processor registers.

## 2.13. Processor Controller

### 2.13.1. Definition

The Processor Controller is a home process that controls the processor when there is no other process to run on it, or when a process has left the processor to enter a waiting queue. There is such a process for each processor.
The basic function of a Processor Controller is to multiplex the processor between the processes waiting for it. To do that, it executes the "runnext(q:&processor)" procedure until it finds a process waiting in the list of ready processes. When it has found such a process, it gives the control of the processor to it.
The activity of the Processor Controller can be defined as follows :

```
' Processor Controller '
cycle
    runnext(processor)
end
```

### 2.13.2. Procedure runnext(q:&processor) (see Algorithm 8)

This procedure is executed by the Processor Controller assigned to the processor q.
First, it transfers the value "idle" to the "running" field defined in the processor record. This value means that no user process is executing instructions on the processor. Then, it examines the head pointer of the ready list, until it finds that the ready queue is not empty. This test is done without disabling the interrupts on that processor, so that the interrupt signals can be received and treated by the home processes. When it finds that the readyqueue is no more empty, then it disables the interrupts and enters a critical region by executing a TAS instruction on the readyprotect boolean variable, defined in the processor record. Then, it selects the most priority process from the ready bit, leaves the critical region and tests the stopwaiting list in the selected process record. If the bit is ON, this means that the selected process is waiting to be stopped but that it may still be inside

a critical region. This is indicated by the "regions" field in the process record. If the value contained in that field is positive, this means that the process is still inside a critical region and that is must complete its activity inside that region before it can be stopped. Thus, control of the processor can be given to it.

If the candidate process is waiting to be stopped but is not inside a critical region, then the processor controller will not give the processor control to that process. Instead, it will put the process in the stopped state and switch off the stopwaiting bit. This completes a stop operation initiated on that process by another process while the prior was in the ready queue.

The processor controller will then enable the interrupts and begin a new cycle of execution to search another candidate.

If the process is allowed to run, then its record address is transferred into the "running" field defined in the processor record. Then, the processor controller gives the value "running" to the state of the process and transfers the context of that selected process into the processor registers. Finally, it enables the interrupts and gives the control of the processor to the selected process.

ALGORITHM 8

```
procedure runnext(q:&processor);
var candidate:&process;
begin
   with q& do
   running:=idle;

   S:while empty(readyqueue)do S;

   disableinterrupts;
   tas(readyprotect);

   Select(candidate,readyqueue);

   readyprotect:=0;
   with candidate& do
   tas(protect);

   if not stopwaiting or regions > 0
      then begin 'run candidate'
             q&.running:=candidate;
             state:=running;
             restorecontext(candidate);
             protect:=0;
             enableinterrupts;
             'The candidate has control of the processor'
           end
      else begin 'stop candidate'
             state:=stopped;
             stopwaiting:=0;
             protect:=0;
             enableinterrupts;
             'Processor controller will try another process'
           end
```

## 2.14. Synchronization

### Utility routines

The following routines will be called by the synchronization primitives to change the state of a process.

### 2.14.1. Procedure blocksem(s:&semaphore)

This procedure is used to block the running process on a given semaphore. First, it puts the process into the semaphore queue. Then, it gives, the value "blocked" to the state field defined in the process record. After that it transfers the semaphore record address and the semtype code to the waitingqueue field of that record. It terminates by saving the calling process context into the context field of the process record.

ALGORITHM 9

```
procedure blocksem(s:&semaphore);
'This procedure is executed on processor Q'
begin
   put(Q&.running,s&.semqueue);
   with (Q&.running)& do
         tas(protect)
         state:=blocked;
         waitingqueue.tag:=semtype;
         waitingqueue.sema:=s;
         savecontext;
         protect:=0
end
```

### 2.14.2. Procedure blockevent(ê:&event)

This procedure is similar to the blocksem procedure but it blocks the running process on a given event rather than on a semaphore.(see Algorithm 10)

ALGORITHM 10

```
procedure blockevent(e:&event);
'This procedure is executed on processor Q'.
begin
   enter(Q&.running,e&.eventqueue);
   with (Q&.running)& do
         tas(protect);
         state:=blocked;
         waitingqueue.tag:=eventtype;
         waitingqueue.evt.=e;
         savecontext;
         protect:=0
end
```

2.14.3. Procedure awake(p:&process)

This procedure inserts process p at the beginning of the ready queue associated to the processor on which the process must run. The head pointer of the ready list is situated in the processor record pointed to by the field "processor" of the process record. The procedure also transfers the value "ready" to the state field of that record. The ready list and the process record are protected inside critical regions.

ALGORITHM 11

```
procedure awake(p:&process);
begin
   with p& do
         tas(protect);
         state:=ready;
         protect:=0;
   with processor& do
         tas(readyprotect);
         enter(p,readyqueue);
         readyprotect:=0
end
```

## 2.15. Semaphores

### 2.15.1. Semaphore description

A semaphore is represented in the common memory by a shared record of type "semaphore" :

```
type semaphore = shared record
                   semprotect : boolean
                   semvalue   : integer
                   semqueue   : &process
                 end.
```

The fields are defined as follows :

semprotect : this field contains a boolean value. It is used to protect the semaphore (by a TAS instruction) against its manipulation by more than one process at the same time.

semvalue : this field contains an integer that represents the semaphore value. It cannot be initialized to a negative value.

semqueue : this field is the head pointer to a list of process records chained together by their "next" fields. There is such a list if the semaphore has a negative value. The number of processes waiting in that process queue is defined by the absolute value of the semaphore. The list is organized as a FIFO queue.

### 2.15.2. Semaphore operations

#### 2.15.2.1. Procedure P(s:&semaphore) (see Algorithm 12)

This procedure implements the P primitive. It disables all interrupts and saves the calling program context in the monitor save area. After that, it executes a TAS instruction on the semprotect field of the semaphore. This allows the process to enter a critical region protected by that variable. Then, it substracts one from the semaphore value. If the result is negative, the procedure inserts the running process at the end of the list of processes waiting on this semaphore, and blocks the process. Blocking a process transfers the monitor

save area to the context field of the process description record. It leaves then the critical region by giving the value 0 to semprotect and enables the interrupts before giving the control of the processor directly to the processor controller.
If the result is nonnegative, the process leaves the critical region, restores back the initial context, enables the interrupts and returns to the calling program.

ALGORITHM 12

```
procedure P(s:&semaphore)
begin
   disableinterrupts;
   saveregisters;
   with s& do
        tas(semprotect);
        semvalue:=semvalue-1;
        if semvalue < 0
        then  begin
              blocksem(Q&.running,s)
              semprotect:=0;
              enableinterrupts;
              runnext(Q)
              end
        else  begin
              semprotect:=0;
              restoreregisters;
              enableinterrupts;
              end
        end
```

2.15.2.2. Procedure V(s:&semaphore) (see Algorithm 13)

This procedure implements the V primitive.After having disabled the interrupts, it enters a critical region protected by semprotect and adds 1 to the value of the semaphore. If the resulting value is not greater than zero, this means that there are processes waiting in the semaphore process queue. The first process is removed from the list by a get operation and is made ready by entering it in the ready queue associated to the processor that has been assigned to it.

Then, the calling process leaves the critical region, enables the interrupts and returns to the calling program. If the resulting value of the semaphore is positive, this means that no process is waiting on the semaphore ; so that the process can leave immediately the critical region, enable the interrupts and return to the calling program

ALGORITHM 13

```
procedure V(s:&semaphore);
var candidate:&process
begin
   disableinterrupts;
   with s& do
   begin
      tas(semprotect);
      semvalue:=semvalue+1;
      if semvalue ≤ 0
         then  begin
                  get(candidate,semqueue);
                  awake(candidate)
               end;
      semprotect:=0;
      enableinterrupts;
   end
```

## 2.16. EVENTS

### 2.16.1. Event description

An event is represented in the common memory by a shared record of type "event" :

```
type event = record
                eventprotect : boolean ;
                eventvalue   : boolean ;
                eventqueue   : &process
             end
```

The fields are defined as follows :

eventprotect : This field contains a boolean value. It is used to protect the event (by a TAS instruction) against its manipulation by more than one process at the same time.

eventvalue : This field contains a boolean. If it is true, this means that an occurrence of the event has occurred.

eventqueue : This field is the head pointer of a list of processes waiting for the next arrival of the event. When such an event occurs, all the processes waiting for it are awakened.

### 2.16.2. Event operations

We define three operations on events :

#### 2.16.2.1. procedure wait(e:&event) (see Algorithm 14)

This procedure allows a process to test if an event has occurred. If it has, the process continues its execution but does not switch off the event, so that the same occurrence of the event can be used by the other processes. If the event did not occur, the process is inserted at the beginning of the list of processes waiting on this event, and blocked before control of the processor is given back to the processor controller.

ALGORITHM 14

```
procedure wait(e:&event);
begin
   disableinterrupts;
   saveregisters;
   with e& do
   begin
      tas(eventprotect)

      if not eventvalue
           then begin
                 blockevent(Q&.running,eventqueue);
                 eventprotect:=0;
                 enableinterrupts;
                 runnext(Q)
                end
           else begin
                 eventprotect:=0;
                 restoreregisters;
                 enableinterrupts;
                end
   end
```

2.16.2.2. procedure signal(e:&event) (see Algorithm 16)

This procedure switches on the event and awakes all the processes waiting on it, if there are such processes.

2.16.2.3. procedure resetevent(e:&event)

This procedure allows a process to switch off an event.

ALGORITHM 15

```
procedure resetevent(e:&event)
begin
   e&.eventvalue:=0
end
```

ALGORITHM 16

```
procedure signal(e:&event)
var p:&process;
begin
   disableinterrupts;
   with e& do
      tas(eventprotect);
      eventvalue:=1;
      if not empty(eventqueue)
           then begin
                p:=eventqueue;
                repeat awake(p)
                until p&.next:=nil
                eventqueue:nil
                end
      eventprotect:=0;
      enableinterrupts
  end
```

## 2.17. Mailbox

### 2.17.1. Mailbox description

We assume that interprocess communication will be made through the use of a common mailbox that can be shared by all the processes.
The main element of that mailbox is a buffer which contains a given number of fixed size message frames. A message frame is a record of type "frame" :

```
type  frame = record
                next:&frame;
                contents:message
              end;
```

The fields in that record are defined as follows :

next : This field is a pointer to another record of type frame. It will be used to chain the message frames together.

contents : This field is a storage area which has to contain a record of type message :

```
type message = array 1..L of character;
const L = message length
```

A process must acquire a message frame from the mailbox before it can use it to send a message to another process. It must also restitute to the mailbox the frames received by it. To assure these operations, all free message frames are chained together to form a list which is pointed to by a head pointer named "freelist".
If the free list is empty, this means that there is no more message frame available in the mailbox. To enable a requester process to wait for the liberation of a frame when there is no available message frame in the mailbox, a semaphore "reserve", initialized to the mailbox capacity, that is the number of message frames in the buffer, is assigned to the mailbox.
The mailbox may thus be represented in the common memory by a shared record defined as follows :

```
var mailbox : shared record
                reserve: &semaphore
                mutexfree: &semaphore
                freelist:&buffer;
                buffer: array 1..B of frame
              end

   const B = Maximum number of message frames in the
             buffer.
```

The fields are defined as follows :

reserve : This is a semaphore, initialized to B, the mailbox capacity. Whenever a process attempts to acquire a message frame, it must execute a P operation on that semaphore. If the resulting value is negative, this means that there is no frame available, and the requesting process will be inserted at the end of the queue associated to that semaphore. On the other side, a V operation must be executed on that semaphore every time a buffer is restituted to the pool of free message frames.

mutexfree : This mutual exclusion semaphore is used to protect the free list of message frames against its manipulation by more than one process at the same time. It must be initialized to 1.

freelist : This is the head pointer to the free list of available frames.

buffer : This field contains all the message frames of the mailbox.

## 2.17.2. Operations on the mailbox

### 2.17.2.1. Procedure allocate(x:&frame)

This procedure must be used to allocate an empty message frame to the calling process. If there is such a frame, it returns the address of the first free message frame pointed to by the freelist pointer. If there is not, the process must wait for it on the "reserve" semaphore.

ALGORITHM 17

```
procedure allocate(x:&frame);
begin
   with mailbox do
   begin
     P(reserve);
     enterregion
     P(mutexfree);
     x:=freelist;
     freelist:=x&.next;
     V(mutexfree);
     leaveregion
   end
```

### 2.17.2.2. Procedure free(x:&frame)

This procedure must be executed by a process every time it has read a message from a given frame. The procedure restitutes the frame to the pool of free message frames. The freed frame becomes the first element of the free list.

ALGORITHM 18

```
procedure free(x:&frame);
begin
   with mailbox do
   enterregion
   P(mutexfree);
   x&.next:=freelist;
   freelist:=x
   V(mutexfree);
   leaveregion
end
```

## 2.17.3. Interprocess Communication

### 2.17.3.1. Mailbox use (see producers-consumer scheme)

Every process can use the mailbox to send a message to another process, or to receive a message sent to it by a sender process.

Prior to sending a message to a consumer process, a producer must allocate a free message frame and then, deposit its message into the allocated frame.

To each process is assigned a list of messages frames sent to it by other processes. The head pointer of that list is the "firstmessage" field defined in the process record. Messages are received in the order of their arrival (FIFO queue of messages). The list has a maximum length, defined at the creation of the process. To prevent a producer process to send a message frame when the message queue of the receiver process is "full", the sender process must execute a P operation on the "emptymail" semaphore defined in the receiver process record. This semaphore has been initialized (at process creation) to the receiver message queue capacity, so that the producer would have to wait on that semaphore if the queue is full.

The receiver message queue is protected by a semaphore mutexmail that assures the mutual exclusion of processes that have to use the queue at the same moment.

To prevent a receiver process to receive of message frame from an empty message queue, a semaphore "fullmail" is defined in the process record. This semaphore is initialized to 0. The receiver process will have to execute a P operation on that semaphore each time it wants to receive a message frame from its queue, so that it will block itself if the queue is empty. On the other hand, any producer which has added a message frame to the message queue of the receiver process must execute a V operation on the "fullmail" semaphore of that process

2.17.3.2. Procedure send(x:&frame,p:&process)

This procedure inserts a message frame at the end of the message queue of process p.

ALGORITHM 19

```
procedure send(x:&frame,p:&process);
var q:&process;
begin
   with p& do
   begin
     q:=firstmessage;
     while q&.next >< nil do
        begin q:=q&.next end ;
     q&.next:=x;
     x&.next:=nil
   end
```

2.17.3.3. Procedure receive(x:&frame)

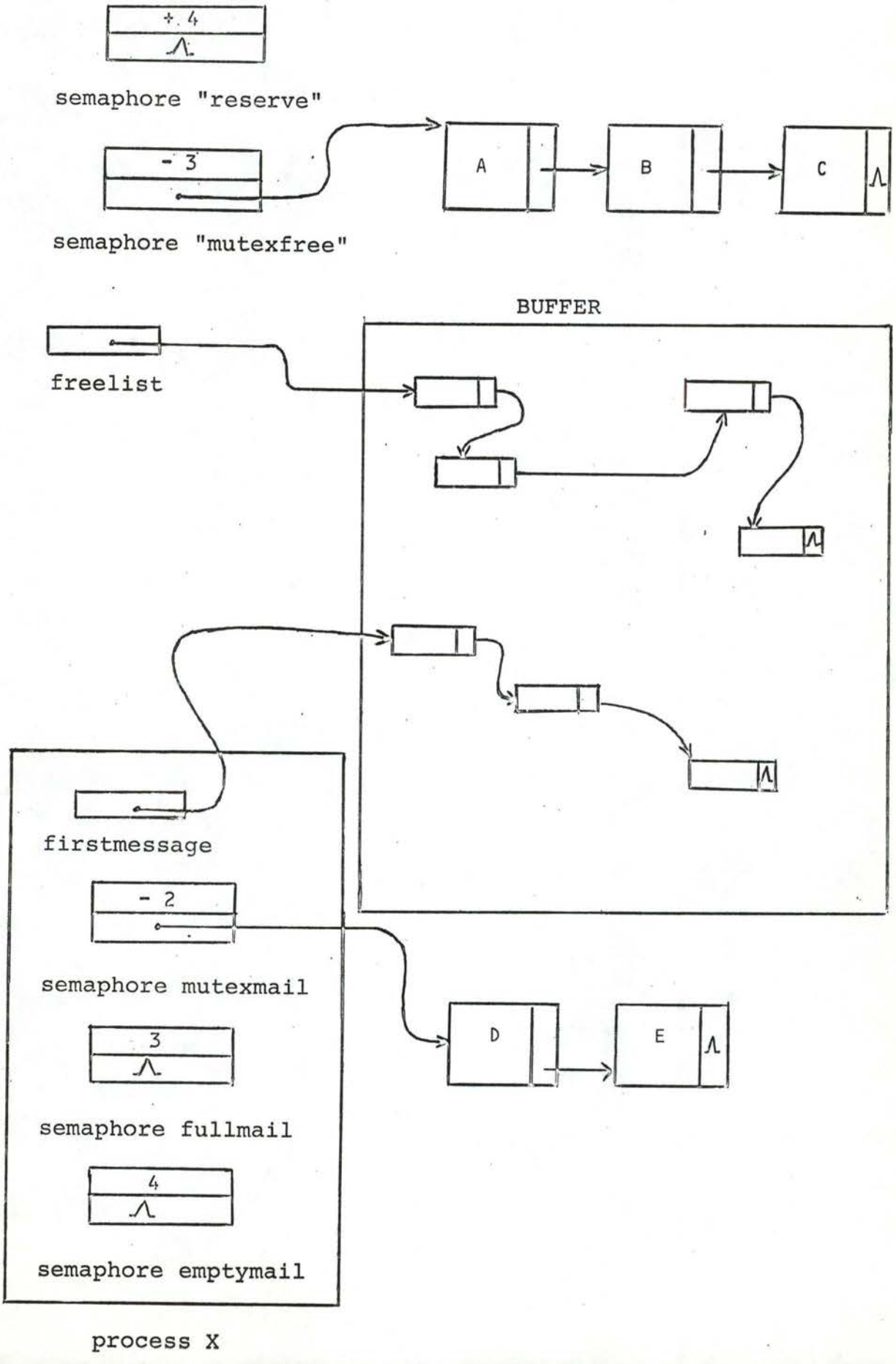
This procedure removes the first message frame from the message queue of the calling process and returns the address of that frame to the process.

ALGORITHM 20

```
procedure receive(x:&frame);
begin
   with (Q&.running)& do
   begin
     x:=firstmessage
     firstmessage:=x&.next
   end
end
```

2.17.3.4. Producers-Consumer Scheme

The following diagram will help us to remember quickly the main elements of the mailbox organization :



In that diagram :

- freelist points to the first element of a linear list of free message frames contained in an area of memory called BUFFER.
- The semaphore "reserve" indicates that four available message frames are chained in the free list.
- Three process - A, B and C - are waiting, on semaphore "mutexfree", for their turn to manipulate the free list (to allocate or free one message frame).

Looking at process X record we see :

- firstmessage points to the first message frame of the process message queue.
- The semaphore "fullmail" indicates that process X message queue contains three message frames.
- Processes D and E are waiting on semaphore "mutexmail" to send a message frame to process X.
- The semaphore "emptymail" indicates that four messages could still be added to process X message queue, prior to blocking the sending processes because the queue is full.

The scheme producers-consumer can be defined as follows (we consider the producer processes that send messages to the consumer X) :

PRODUCERS

CONSUMER X

```
var d,y:&frame ;
var message1,message2:message ;
const Xmax=maximum length of Process X message queue ;
      init  with X& do
            begin
              emptymail&.semvalue:=Xmax;
              mutexmail&.semvalue:=1;
              fullmail&.semvalue:=0;
              firstmessage:=nil
            end
```

```
cycle
   product(message);
   P(X&.emptymail);
   allocate(d);
   d&.contents:=message1;
   with X& do
   P(mutexmail);
   send(d,X);
   V(mutexmail);
   V(fullmail)
end
```

```
cycle
  with X& do
  P(fullmail);
  P(mutexmail);
  receive(y);
  V(mutexmail);
  message2:=y&.contents
  free(y);
  V(emptymail);
  consume(message2)
end
```

## 2.18. Stopping and Starting a process

### 2.18.1. Introduction

A process which is in the stopped state is not runnable. The difference between a stopped process and a blocked process is that the former is not waiting in a semaphore or event queue and cannot therefore be awakened by the V or signal synchronization primitives. The only way to awake a stopped process is by executing the start operation on that process.

The following rules are inherents to the stop operation :

1°) Any user process may be stopped, whatever its state is.

2°) A process cannot enter the stopped state if it is inside a critical region.

3°) The stop operation on a process which is inside a critical region, must be delayed until the process leaves the critical region. If the process is inside nested critical regions,then the stop operation must be delayed until it leaves the first critical region it entered.

4°) A process can be restarted at the point it was stopped.

As consequences of these rules, we say :

a) A running process can be stopped by another process. We assume that this will be done by sending an interrupt signal to the processor on which the process to be stopped is running.

b) If the process to be stopped is blocked in a semaphore or event queue, the V or signal operation must not awake it once it is stopped. This will be assured by taking the process away from this queue.

c) If the stopped process was in the blocked state when it was stopped, it must enter that state again when it is restarted but not if the signal it was waiting for did occur. To resolve that problem, we will force the process, when it is restarted, to execute again the P or wait operation that had blocked it.

### 2.18.2. Summarize of the stop and start operation

The stop and start operations may be summarized as follows (the process to be stopped is called the "candidate").

- We assume that the order of stopping a process is given by a user process, by executing the stop(p) procedure. (see 2.18.3.11)
- If the candidate is not runnable on the same processor as the ordering process, then the latter will send an interrupt signal to the other processor. This interrupt signal will activate a home process that will complete the stop operation, by executing the initiatestop(p) procedure.
- If the candidate is inside a critical region, its stopwaiting bit is set on and the stop operation will be delayed until the process leaves the first critical region it entered.
- If the candidate is blocked, then it is taken away from its waiting queue and stopped. The Program Counter will be modified in the context so that, when it is runnable again, the process will execute the same P or wait operation that caused its blocked state.
- If the candidate is running, it is stopped directly, unless it is inside a critical region.
- If the candidate is ready and not inside a critical region, it will stay in the ready queue but its stopwaiting bit is set on. It will be stopped definitively by the processor controller when selected by it.

- A start operation on a process makes the process ready if it is stopped. If it is not, then it is assumed that it is waiting for it, and the procedure will set off its stopwaiting bit.

The following pages define step by step the mechanism used to stop or start a process.

## 2.18.3. Procedures

### 2.18.3.1. - Procedure takeaway(p:&process,x:headpointer)

This procedure searches a given process p in a process queue pointed to by the given head pointer x. The queue may be a semaphore or event queue. The process is assumed to be in the queue. When the procedure has found the process, it takes it away from the list.

ALGORITHM 21

```
procedure takeaway(p:&process,x:headpointer);
var q:&process;
begin
   if x=p then x:=p&.next
   else  begin
         q:=x;
         while q&.next><p
           do begin q:=q&.next end;
         q&.next:=p&.next
         end
```

### 2.18.3.2. - Procedure modifypc(p:&process,x:blockcode)

This procedure is used to stop a blocked process. It modifies the Program Counter field defined in the context area of the given process record, in such a way that, when the process will run again, it will execute the last P or wait operation that had blocked it. This operation is defined by the block-code given by x :

```
type blockcode = (P,wait)
```

If the block code is P, then the process was blocked by a P operation. Otherwise, it was blocked by a wait operation.

2.18.3.3. - Procedure Vstop(p:&process,s:&semaphore)

This procedure is used to stop a process blocked on a semaphore. It is similar to a V primitive, in that it increments the given semaphore value by one and takes a process away from the semaphore queue. The main difference is that it does not take the first process in the queue but searches the given process in it before taking it away. This is done by calling the takeaway procedure for the given process and the semaphore queue. Another difference is that the process is not made ready but stopped. The context field of the process is also modified so that the program counter will point to the P operation that blocked the process. This will force the process to execute again the P operation when it is restarted.

ALGORITHM 22

```
procedure Vstop(p:&process,s:&semaphore);
begin
   with s& do
   semvalue:=semvalue+1;
   takeaway(p,semqueue);
   with p& do
   state:=stopped;
   modifypc(p,P)
end
```

Note Process p is assumed to be effectively in the given semaphore queue.

2.18.3.4. - Procedure signalstop(p:&process,e:&event)

This procedure is used to stop a process blocked on an event. It takes away the given process from the given event queue, puts it in the stopped state and modifies the Program Counter field in its context so that, when restarted, it will execute again the wait operation that blocked it. (see ALGORITHM 23)

ALGORITHM 23

```
procedure signalstop(p:&process,e:&event);
begin
   with e& do takeaway(p,eventqueue);
   with p& do
   state:=stopped;
   modifypc(p,wait)
end
```

2.18.3.5. - Procedure stopfromblock(p:&process) (see Algorithm 24)

This procedure is used to stop a blocked process. First, it tests if the process is blocked. If it is, then it may be blocked on a semaphore or on an event. This is indicated by the tag field of the waitingqueue record defined in the process record.

- If the process is blocked on a semaphore, then the procedure enters two nested critical regions which protect respectively the semaphore and the process record. Between the time the procedure began to test the process state and the one it finished to enter the critical regions, the process could have been made ready by a V operation executed by a process on a different processor, so that the procedure must test again the state of the process it must stop. If it is still blocked, then a Vstop operation must be executed to put it into the stopped state.
  If not, the procedure can leave immediately the nested critical regions.
- If the process is blocked on an event, the procedure will execute similar operations, but it must stop the process by calling the signalstop procedure rather than executing a Vstop operation.

ALGORITHM 24

```
procedure stopfromblock(p:&process);
begin
   with p& do
   if state=blocked then
   begin
      with waitingqueue do
      if tag=semtype
      then begin
             tas(sema&.semprotect);tas(protect);
             if state><ready
             then begin Vstop(p,sema)end;
              protect:=0;sema&.semprotect:=0
             end
      else begin
             tas(evt&.eventprotect);tas(protect);
             if state><ready
             then begin signalstop(p,evt)end;
              protect:=0;evt&.eventprotect:=0
             end
   end
end
```

2.18.3.6. - Procedure initiatestop(p:&process) (see Algorithm 25)

This procedure stops a process or initializes the stopping operation. We assume it is executed on the same processor that the one assigned to the process it must stop.

This procedure first enters a critical region that protects the process record and examines the field "regions" of that record. If it contains a positive value, this means that the process to stop is still inside one or more critical regions. In such a case, it cannot be stopped. The procedure simply switches on the stopwaiting bit of that process and leaves the critical region. The process will be stopped later, when it leaves the first critical region it entered, thus when "regions" will take the value 0.

- The same is done if the process is in the ready state. The stopwaiting bit is set on and the procedure leaves the critical region. The stopping will be delayed until the processor controller selects the process as a candidate to run. If the candidate has its stopwaiting bit set on, the processor controller will stop it instead of giving the control of the processor to it.
- If the process to stop is still in the running state, then the procedure will save its context and put it into the stopped state. It then leaves the critical region and returns.
- If the process is in the blocked state, then the procedure leaves the critical region of the process record before it calls the "stopfromblock" procedure which will try to stop the process. The procedure has to leave first the critical region, to respect the hierarchal rules that allow deadlocks to be avoided. A consequence is that another process may now enter the critical region and make the process ready. In such a case the procedure will have to try again the stop operation and it will do so until the process is stopped or its stopwaiting bit is on.

2.18.3.7. - Procedure enterregion (see Algorithm 26)

This procedure is used by a running process when it enters a critical region. It adds 1 to the "regions"field defined in its process record, so that the process cannot be stopped until it has left the critical region

2.18.3.8. - Procedure leaveregion (see Algorithm 27)

This procedure is executed by a running process when it leaves a critical region. It substracts one from the "regions"field defined in its process record.
If the new value is zero and the stopwaiting bit of that process is on, then the process must be stopped. The procedure does that before giving the control of the processor to the processor controller

ALGORITHM 25

```
procedure initiatestop(p:&process);
begin
   with p& do
   repeat
      tas(protect);
      if regions>0 or state=ready
      then begin 'do not stop now'
           stopwaiting:=1;
           protect:=0;
           end
      else begin
           if state=running
           then begin 'stop runningprocess'
                savecontext
                state:=stopped;
                protect:=0
               end
           else begin 'try to stop blocked process'
                protect:=0 'to avoid deadlock'
                stopfromblock(p)
               end
   until stopwaiting or state=stopped
```

ALGORITHM 26

```
   procedure enterregion
   'This procedure is executed on processor Q'
   begin
      disableinterrupts
      with (Q&.running)& do
      tas(protect);
      regions:=regions+1;
      protect:=0;
      enableinterrupts
   end
```

ALGORITHM 27

```
procedure leaveregion
'This procedure is executed on processor Q'
begin
   disableinterrupts;
   with (Q&.running)& do
      tas(protect);
      regions:=regions=1;
      if stopwaiting and regions=0
      then begin
           savecontext;
           state:=stopped;
           protect:=0;
           enableinterrupts;
           runnext(Q)
           end
      else begin
           protect:=0;
           enableinterrupts
           end
end
```

2.18.3.9. - Procedure interrupton(q:&processor,p:&process,sending: &process)

This procedure is used to stop a process p which is not runnable on the same processor as the calling process. Its effects is to send an interrupt signal to the processor on which process p is runnable. This interrupt signal, when received, will give control of that processor to a home process which will first execute a V(stopper) operation to awake the sending process. The home process will then initiate the stopping of process p.

2.18.3.10. - Procedure interruptoff(q:&processor)

This procedure is used to set off the interrupt line set on the "interrupton" procedure.

2.18.3.11 - Procedure stop(p:&process) (see Algorithm 29)

This procedure is used by a user process to stop another process. It first disables the interrupts and tests if the processor on which the given process is runnable is the same that the one on which the procedure is executed. If it

is, then it stops the process p or, at least, initiates its stopping. This is done by calling the initiatestop(p) procedure.

If process p is not runnable on the same processor, then the running process enters a critical region and sends an interrupt signal to the processor assigned to process p, by executing the "interrupton" procedure.Then, the calling process blocks itself on its private "stopper" semaphore, defined in its process record. It will be awakened by the home process which must receive the interrupt signal and that will stop or initiate the stopping of process p. Once awakened, the stopping process will set off the interrupt line that served to send the interrupt signal, and leave the critical region.

2.18.3.12.- Procedure start(p:&process)

This procedure is used to awake a stopped process. If the process is really stopped then it puts it into the ready state by executing the awake(p) procedure. It it is not, then it simply switches off the stopwaiting bit of that process.

```
ALGORITHM 28
    procedure start(p:&process)
    begin
       with p& do
       if state=stopped then awake(p)
          else begin
                  tas(protect);
                  stopwaiting:=0;
                  protect:=0
               end
    end
```

ALGORITHM 29

```
procedure stop(p:&process);
'This procedure is executed on processor Q'
begin
   disableinterrupts;saveregisters;
   with p& do
      if processor><Q
         then begin 'process p is not runnable on
                                          processor Q'
                enterregion;
                interrupton(processor,p,Q&.running);
                enableinterrupts;
                P((Q&.running)&.stopper);
                interruptoff(processor);
                leaveregion
              end
         else begin
                initiatestop(p);
                if state=stopped
                   then begin
                          enableinterrupts;
                          runnext(Q)
                          end
                   else begin
                          restoreregisters;
                          enableinterrupts
                          end
end
```

## 2.19. Peripherals

### 2.19.1. Peripheral description

A peripheral is represented in the common memory by a shared record of type "peripheral" :

```
type peripheral =
     shared record
                mutex:&semaphore;
                reserved:boolean;
                next:&peripheral;
                requesters:&process;
                peripheraluser:&semaphore;
                parameters:&paramtable
            end
```

The fields are defined as follows :

mutex : This is a mutual exclusion semaphore used to protect the peripheral record against its manipulation by more than one process at a time.

reserved : This boolean is "true" when the peripheral has been reserved by a process for its exclusive use.

next : This field is used to link together all the peripherals that have been acquired by a process. The head pointer of such a list is defined by the field "requestqueue" located in the requester process record.

requesters : This field is used to link together all the processes that have requested the peripheral but could not be satisfied because the peripheral had been acquired by another process.

peripheraluser : This field points to a semaphore on which the peripheral user can wait by executing a P operation on it. It will be awakened by a V operation executed by the homeprocess which terminates the IO.

parameters : This field points to a table which contains parameters used by the start or terminate IO routines.

## 2.19.2. Peripheral Requests and Releases

### 2.19.2.1. Introduction

We consider the peripherals that must be "requested" prior to being used by a process and "released" by that process before they can be used by other requesting processes.

We assume that the peripheral are grouped in hierarchal classes but that a class contains only one peripheral. However, no control is done, at the monitor level, in order to verify if a request ,respects the hierarchal order defined to prevent deadlock. Such control is assumed to be done at compilation time.

### 2.19.2.2. Objective

When many processes are requesting the use of a peripheral at the same moment, only one of them may be satisfied in its request. The other requester processes must be delayed until the peripheral is freed again by a release operation executed by the process which acquired the peripheral.

A solution is to assign a semaphore to each peripheral and initialize it to 1. The request procedure would force the requester process to execute a P primitive on that semaphore so that the process would block itself and enter the semaphore queue if the resulting semaphore value is negative. The process would get control of the peripheral when a V operation is executed on that semaphore by the release procedure called by the process which acquired the peripheral.

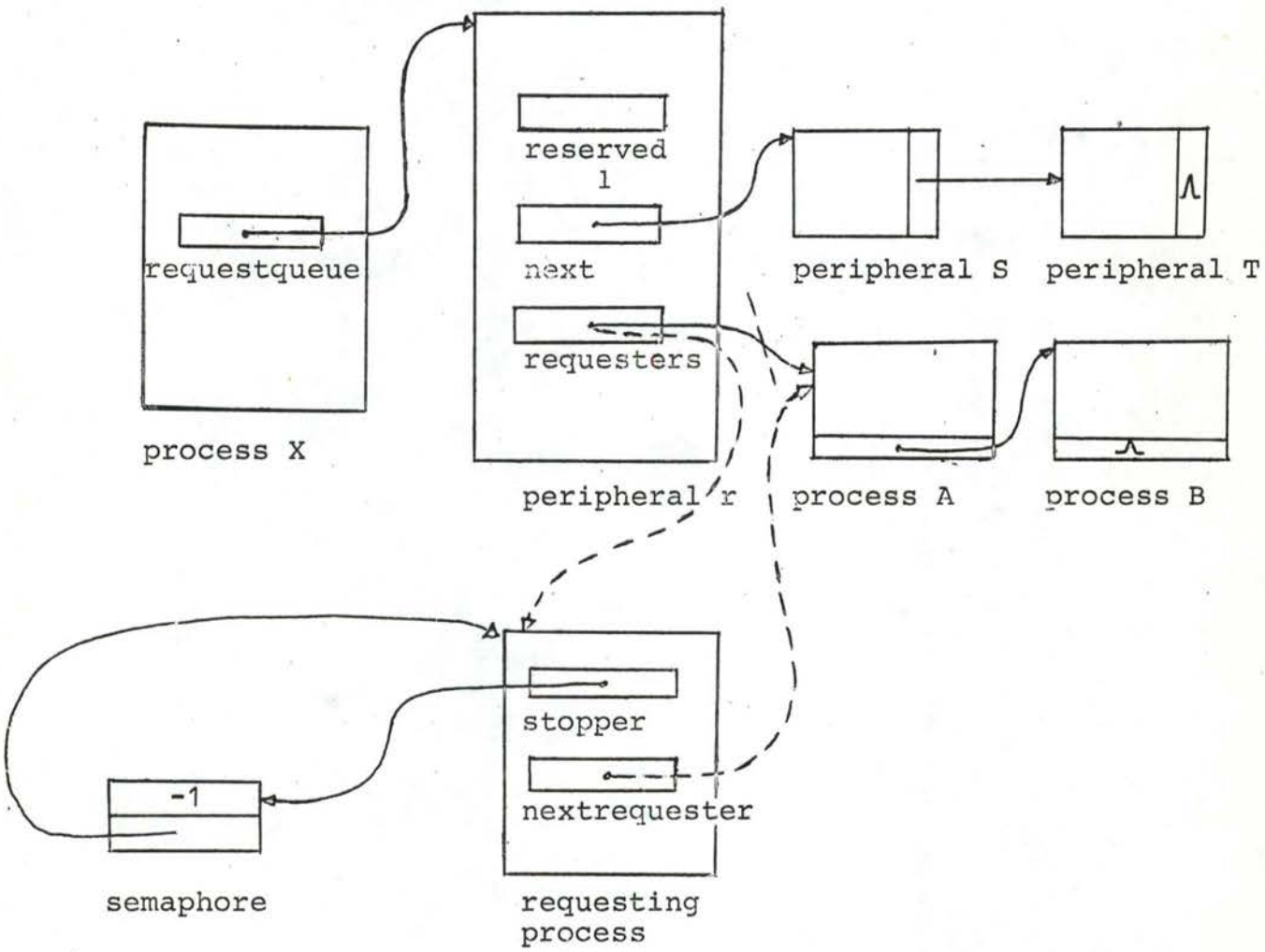
The inconvenience of such a solution is that the P and V primitives schedule the waiting processes in the order of their arrival (FIFO). However, we could wish to schedule them in a different order,for example, in function of their priorities. To do so, we could create a special V primitive that would select the most priority process from a semaphore queue. Another solution would be to use the standard P and V primitives and create an artifice that will allow us to schedule the waiting processes in their priority order. We choose that solution.

Therefore, the following procedures may be regarded as an example which shows how to use the standard synchronization primitives P and V to schedule a queue of waiting processes, in a different order than FIFO.

2.19.2.3. SCHEME

---

Let us consider the following scheme :




On this scheme, we see that the requested peripheral (r) has been acquired by process X and that processes A and B are waiting in the requesters queue pointed by the "requesters" field of peripheral r.
Peripherals S and T have also been acquired by process X and are linked to peripheral r to form the requested peripherals queue pointed by the "requestqueue" pointer defined in the process X record.

The boolean "reserved" has been set to "1" to indicate that the peripheral has been reserved.

Let us now consider the requesting process at the moment it is executing the request procedure.
This process tests the "reserved" field to verify if the peripheral has been reserved or not. If it has, then it must enter the requesters queue and wait ; that is, it must block itself. It will do it by executing a P primitive on its private semaphore "stopper", defined in its process record.

2.19.2.4. Procedure request(r:&peripheral) (see Algorithm 30)

This procedure must be called by every process which has to request a peripheral before using it. It tests if the reserved field is set to "1" : If it is, then the peripheral has been reserved by another process and the calling process must wait. The procedure links that process in the peripheral requesters queue and forces the process to block on its "stopper" semaphore. If the peripheral is free, then the "reserved" field is set to "1" and the peripheral record is put on the top of the process requested peripherals queue defined in its process record (by the head pointer "request-queue"), and a V operation is executed on the "stopper" semaphore of that process, so that it will not block itself when it executes the P operation which follows.

2.19.2.5. Procedure release (see Algorithm 31)

This procedure is used to release the last peripheral that has been acquired by the calling process. Thus, it assumes that peripheral requests were done in the hierarchal order that avoid deadlock and that there is one peripheral by class, only.

ALGORITHM 30

```
procedure request(r:&peripheral);
begin
  disableinterrupts;enterregion;
  with r& do
  if not reserved then
     begin
       reserved:=1;
       with(Q&.running)& do
            r&.next:=requestqueue;
            requestqueue:=r;
            V(stopper)
     end
  else enter(Q&.running,requesters);
  V(mutex);
  leaveregion;enableinterrupts;
  with (Q&.running)& do
  P(stopper)
end
```

The procedure takes the peripheral record away from the list pointed by the head pointer "requestqueue" defined in the calling process record. Then it tests if other processes are waiting for the peripheral. If there are, then the procedure selects the most priority of them by calling the "takeoff" procedure, and awakes it by executing a V operation on the "stopper" semaphore of that process.

If no process is waiting in the queue, then the procedure switches off the boolean "reserved" defined in the peripheral record.

ALGORITHM 31

```
procedure release ;
var r:&peripheral;candidate:&process;
begin
  disableinterrupts;enterregion;
  with (Q&.running)& do
  r:=requestqueue
  requestqueue:=r&.next;
  with r& do
  P(mutex);
  if requesters = nil then reserved:=0
     else begin
            takeoff(candidate,requesters);
            V(candidate&.stopper);
          end
  V(mutex);
  leaveregion;enableinterrupts;
end
```

# C O N C L U S I O N

The Multiprocessor System designer must establish a set of rules, the purpose of which is to co-ordinate and control the activities of the processes which have to work on the system.
These rules are materialized by a set of data and procedures, called a "Monitor".

This thesis allowed me to reach two objectives :

- The acquisition of a theoretical knowledge of the main concepts inherent to the multiprocesses environments.
- The consolidation of that knowledge by defining the kernel of a real-time monitor for multiple processor microcomputer systems.

To achieve these purposes, an important bibliographical work has been required. From that study appeared the main concepts of multiprocess environments. These concepts may be summarized as follows :

- processes and states of processes
- process synchronization
- mutual exclusion
- deadlock
- interprocess communication.

They were used to define the kernel in the following way :

- A process is represented by a record which contains all information needed by the monitor procedures to control its activity.
- A process can have one of the following states: ready, running, blocked or stopped.
  The rules that allow a process to pass from one state to another are defined in a State Diagram.

- Synchronization of processes is done through the use of semaphores or events.
- Mutual exclusion is resolved by the TAS instruction implementation and by its use to define critical regions. Mutual exclusion semaphores can also be used.
- Deadlock is prevented by a hierarchal allocation of resources. However, the monitor does not check if the hierarchal order of requests is respected. That control must be done at compilation time.
- Interprocess communication is realized through the use of a common mailbox.

The kernel has the following limits :

- It has been defined to be implemented on a model of configuration proposed by the most important microprocessor constructors.
- The number of processes if fixed.
- The programs are fixed, once and for all, in private ROM memories.
- Synchronization and communication are done through the common memory.
- A process can execute its program on one processor only.
- Protection of objects and deadlock prevention problems are essentially resolved at compilation time.

The monitor has been defined in a modular fashion, in such a way that it is easy to modify it with regard to the applications. So that, we are able to adapt a monitor to the application that the system must process.

# B I B L I O G R A P H Y

(A&R) G. ADAMS, T. ROLANDER (Intel Corporation) :
" *Design Motivations for Multiple Processor Micro-computer Systems* "
- Computer Design, March 1978.

J.S. BANINO :
" L'accès aux informations dans le système MULTICS "
- IRIA Laboria, laboratoire de recherche en informatique et automatique / rapport de recherche n° 217, Févr. 1977.

C. GORDON BELL, PETER FREEMAN :
" C.ai - A computer architecture for AI Research "
- Fall Joint Computer Conference, 1972 - pp. 779-790.

A.J. BERNSTEIN, G.D. DETLEFSEN & R.H. KERR :
" Process Control and Communication "
- ACM/2d Symposium on Operating System Principles, Oct. 20-22, 1969 - Princeton University. (NY ; Association for Computing Machinery, 1969), pp. 60-66.

(BH1) PER BRINCH HANSEN :
" *The Nucleus of a Multiprogramming System* "
- CACM, vol. 13, n° 4, April 1970.

PER BRINCH HANSEN :
" RC 4000 Software Multiprogramming System ",
- A/S Regnecentralen, Copenhagen, Feb. 1971.

PER BRINCH HANSEN :
" Structured Multiprogramming "
- CACM, July 1972, Vol. 15, n° 7.

PER BRINCH HANSEN :
" Concurrent programming concepts "
- Information Science, California Institute of Technology, Feb. 1973.

(BH2) PER BRINCH HANSEN :
" *Operating System Principles* "
- Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

PER BRINCH HANSEN :
" Concurrent Pascal Report "
- Information Science, California Institute of Technology, June, 1975.

PER BRINCH HANSEN :
" The Solo Operating System "
- Information Science, California Institute of Technology, July, 1975.

PER BRINCH HANSEN :
" Concurrent Pascal Machine "
- Information Science, California Institute of Technology, Oct., 1975.

PER BRINCH HANSEN :
" A Real-Time Scheduler "
- Information Science, California Institute of Technology, California 91125, Nov., 1975.

(BH3) PER BRINCH HANSEN :
" *The Architecture of concurrent Programs* "
- Prentice-Hall, Inc. Englewood Cliffs, New Jersey 07632, 1977.

C. BéTOURNé, J. BOULENGER, J. FERRIé, C. KAISER, J. KOTT, S. KRAKOWIAK, J. MOSSIèRE :
" Process Management and Resource Sharing in the Multiaccess System 'ESOPE'".
- ACM/2d Symposium on O.S. Principles, Oct. 20-22, 1969 - Princeton University (NY : Association for Computing Machinery, 1969) pp. 67-74.

F.C. COLON, R.M. GLORIOSO, W.H. KOHLER, D.W. LI :
" Coupling small computers for performance enhancement "
- National Computer Conference, 1976, pp. 755-764.

(Cr) A. CROCUS (nom collectif) :
" *Systèmes d'exploitation des ordinateurs* "
- Dunod, 1975.

P. DENNING :
" *Resource allocation in Multiprocess computer Systems* "
- Massachussets Institute of Technology, Project MAC. (thesis May 1968).

(D) EDSGER W. DIJKSTRA :
" *The structure of the "THE" - Multiprogramming System* "
- CACM, vol. 11, n° 5, May 1968.

PHILIP H. ENSLOW Jr. :
" Multiprocessor Organization - A Survey "
- ACM Computing Surveys : special issue : parallel processors and Processing, vol. 9, n° 1, March 1977, pp. 103-129.

J. FERRIé :
" Contrôle de l'accès aux objets dans les systèmes informatiques "
- Thèse de doctorat d'état es-sciences mathématiques, présentée à l'université de Paris, 29 sept. 1975.

J. FERRIé / D. LANCIAUX :
" Le Système Plessey 250 "
- IRIA - rapport de recherche n° 168 - Avril 1976.

CLAUDE GASCHET :
" Les systèmes multiprocesseurs de National Semiconducteur ".
- Minis et Micros n° 68 (17 mars 1978).

DOMINIQUE GIROD :
" Un monochip 16 bits : le TMS 9940 "
- Minis et Micros n° 73 (26 mai 1978), n° 74 (9 juin 1978) et n° 75 (23 juin 1978).

BERNARD GIRARD & GILLES MICHEL :
" Les langages évolués pour le temps réel "
- Minis et Micros n° 55 et 62, 9 sept. 1977 et 16 décembre 1977.

ROBERT M. GRAHAM :
" Protection in an Information Processing Utility "
- CACM, vol. 11, n° 5, May 1968, pp. 365-369.

(Ha) A. NICO HABERMANN :
" *Synchronization of Communicating Processes* "
- CACM, vol. 15, n° 3, March 1972, pp. 171-176.

M. HANNE :
" Micros et Moniteurs Temps Réel "
- Minis et Micros n° 63 (6 janv. 1978), n° 65 (3 fév. 1978) et n° 66 (17 fév. 1978).

E.E. HEART, S.M. ORNSTEIN, W.R. CROWTHER, W.B. BARKER :
" A new minicomputer/multiprocessor for the ARPA network"
- National Computer Conference, 1973 - pp. 529-537.

C.A.R. HOARE :
" Monitors : An Operating System Structuring Concept "
- CACM, Oct. 1974, vol. 17, n° 10.

P. HUGOT :
" Minis et Moniteurs Temps Réel "
- Minis et Micros n° 44, 45 et 48, Mars et Avril 1977.

KATHLEEN JENSEN & NIKLAUS WIRTH :
" Pascal, User Manual and Report " ; Lecture Notes in Computer Science, edited by G. Groos and J. Hartmanis ;
Springer-Verlag, Berlin. Heidelberg. New York 1976.

ANITA K. JONES, R.J. CHANSLER, Jr., IVOR DURHAM :
" Software management of Cm* - a distributed multiprocessor"
- National Computer Conference, 1977, pp. 657-663.

DONALD E. KNUTH :
" The Art of Computer Programming "
vol. 1/Fundamental Algorithms.
- Addison - Wesley Publishing Company, 1968.

(La1) B.W. LAMPSON :
" Scheduling and Protection in interactive Multiprocessor Systems " (thesis)
- (University of California, Berkeley)
. Doc. n° P11, January 20, 1967 - Office of the Secretary of Defense, Advanced Research Projects Agency, Washington 25, D.C.

(La2) B.W. LAMPSON :
" *A Scheduling Philosophy for Multiprocessing Systems* "
- CACM, vol. 11, n° 5, May 1968, pp. 347-360.

B.W. LAMPSON :
" Dynamic protection structures "
(-Berkeley Computer Corporation, Berkeley, California)
Proc. AFIPS 1969, FJCC 35, AFIPS Press, Montvale, N.J., 1969, pp. 27-38.

(M&D) STUART E. MADNICK & JOHN J. DONOVAN :
" *Operating Systems* "
- McGraw - Hill Book Company, 1974.

J.C. MASSON :
" Le Multitraitement avec le microprocesseur SC/MP : évaluation de performances selon le nombre de microprocesseurs connectés à un unique bus"
- Electronique et Applications industrielles, n° 249, 15 mars 1978.

J.CL. MATHON :
" Un micro-ordinateur conçu pour la réalisation de systèmes multiprocesseurs " (TMS 9940)
- Electronique et Applications industrielles, n° 255, 15 juin 1978.

K. NOGUCHI, J. OHNISHI, H. MORITA :
" Design consideration for a heterogeneous tightly-coupled multiprocessor system "
- National Computer Conference, 1975 - pp. 561-565.

ELLIOT I. ORGANICK :
" The MULTICS system : an examination of its structure "
- The MIT Press (1972).

JANAK PATHAK :
" Software setup eases traffic flow for multiprocessors "
(National Semiconductor Corp.)
- Electronics, March 31, 1977.

S.M. ORNSTEIN, W.R. CROWTHER, M.F. KRALEY, R.D. BRESSLER, A. MICHEL :
" Pluribus - A reliable multiprocessor "
- National Computer Conference, 1975, pp. 551-559.

ADAM OSBORNE :
" An introduction to microcomputers "
vol. 1 : Basic concepts
vol. 2 : Some Real Products, June 1977.
- Adam Osborne and Associates, Incorporated P.O. Box 2036, Berkeley California 94702.

D.L. PARNAS :
" A Technique for Software Module Specification, with examples "
- CACM, May 1972, vol. 15, n° 5, pp. 330-336.

D.L. PARNAS :
" On the criteria to be used in decomposing systems into Modules "
- CACM, Dec. 1972, vol. 15, n° 12, pp. 1053-1058.

R2E :
" Micral S - Moniteur Temps Réel - MTR S"
- Réalisations Etudes Electroniques, réf. L23-F, Jan 1977.

(Sa) J.H. SALTZER :
" *Traffic Control in a multiplexed computer System* "
MAC-TR-30, thesis, July 1966.
- Project MAC, Massachussetts Institute of Technology, 545 Technology Square, Cambridge, Massachussetts, July 1966.

MICHEL J. SPIER & ELLIOT I. ORGANICK :
" The Multics Interprocess Communication facility "
- ACM/2d Symposium on operating systems principles, October 20-22, 1969 - Princeton University. (New York : Association for computing Machinery, 1969) pp. 83-91.

J.E. STOCKENBERG, P.C. ANAGNOSTOPOULOS, R.E. JOHNSON, R.G. MUNCK, G.M. STABLER, A. VAN DAM :
" Operating system design considerations for micro-programmed mini-computer satellite Systems "
- National Computer Conference, 1973, pp. 555-562.

R.J. SWAN, S.H. FULLER, D.P. SIEWIOREK :
" Cm* - A modular, multi-microprocessor "
- National Computer Conference, 1977 - pp. 637-644.

R.J. SWAN, A. BECHTOLSHEIM, KWOK-woonkai, J.K. OUSTERHOUT :
" The implementation of the Cm* multi-microprocessor "
- National Computer Conference, 1977, pp. 645-655.

C.K. TANG :
" Cache system design in the tightly coupled multi-processor system "
- National Computer Conference, 1976 - pp. 749-753.

N. WILHELM, D. PESSEL, C. MERRIAM :
" The CERF Computer System "
- National Computer Conference, 1976, pp. 765-768.

W.A. WULF, D.B. RUSSEL, A.N. HABERMANN :
" BLISS : A Language for Systems Programming "
- CACM, Dec. 1971, vol. 14, n° 12, pp. 780-790.

W.A. WULF, C.G. BELL :
" C.mmp - A multi-mini-processor "
- Fall Joint Computer Conference, 1972, pp. 765-777.

W.A. WULF :
" Hydra : the kernel of a Multiprocessor System "
- Communication of the ACM, vol. 17, n° 6, June 1974. pp. 337-345.